Chapter 4. Getting Help in Red Hat Enterprise Linux

[**Reading Manual Pages**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch04/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Guided Exercise: Reading Manual Pages**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch04s02/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Reading Info Documentation**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch04s03/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Guided Exercise: Reading Info Documentation**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch04s04/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Lab: Getting Help in Red Hat Enterprise Linux**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch04s05/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Summary**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch04s06/9a2ef70f-4e72-42df-a498-b694b274af27)

**Abstract**

|  |  |
| --- | --- |
| **Goal** | Resolve problems by using local help systems. |
| **Objectives** | * Find information in local Linux system manual pages. * Find information from local documentation in GNU Info. |
| **Sections** | * Reading Manual Pages (and Guided Exercise) * Reading Info Documentation (and Guided Exercise) |
| **Lab** | Getting Help in Red Hat Enterprise Linux |

Reading Manual Pages

Objectives

After completing this section, you will be able to find information in local Linux system manual pages.

Introducing the man command

One source of documentation that is generally available on the local system are system manual pages or *man pages* These pages are shipped as part of the software packages for which they provide documentation, and can be accessed from the command line by using the **man** command.

The historical Linux Programmer's Manual, from which man pages originate, was large enough to be multiple printed sections. Each section contains information about a particular topic.

**Table 4.1. Common Sections of the Linux Manual**

| **Section** | **Content type** |
| --- | --- |
| 1 | User commands *(both executable and shell programs)* |
| 2 | System calls *(kernel routines invoked from user space)* |
| 3 | Library functions *(provided by program libraries)* |
| 4 | Special files *(such as device files)* |
| 5 | File formats *(for many configuration files and structures)* |
| 6 | Games *(historical section for amusing programs)* |
| 7 | Conventions, standards, and miscellaneous *(protocols, file systems)* |
| 8 | System administration and privileged commands *(maintenance tasks)* |
| 9 | Linux kernel API *(internal kernel calls)* |

To distinguish identical topic names in different sections, man page references include the section number in parentheses after the topic. For example, **passwd**(1) describes the command to change passwords, while **passwd**(5) explains the /etc/passwd file format for storing local user accounts.

To read specific man pages, use **man *topic***. Contents are displayed one screen at a time. The **man** command searches manual sections in alphanumeric order. For example, **man passwd** displays passwd(1) by default. To display the man page topic from a specific section, include the section number argument: **man 5 passwd** displays passwd(5).

Navigate and Search Man Pages

The ability to efficiently search for topics and navigate man pages is a critical administration skill. GUI tools make it easy to configure common system resources, but using the command-line interface is still more efficient. To effectively navigate the command line, you must be able to find the information you need in man pages.

The following table lists basic navigation commands when viewing man pages:

**Table 4.2. Navigating Man Pages**

| **Command** | **Result** |
| --- | --- |
| **Spacebar** | Scroll forward (down) one screen |
| **PageDown** | Scroll forward (down) one screen |
| **PageUp** | Scroll backward (up) one screen |
| **DownArrow** | Scroll forward (down) one line |
| **UpArrow** | Scroll backward (up) one line |
| **D** | Scroll forward (down) one half-screen |
| **U** | Scroll backward (up) one half-screen |
| **/***string* | Search forward (down) for *string* in the man page |
| **N** | Repeat previous search forward (down) in the man page |
| **Shift**+**N** | Repeat previous search backward (up) in the man page |
| **G** | Go to start of the man page. |
| **Shift**+**G** | Go to end of the man page. |
| **Q** | Exit **man** and return to the command shell prompt |

Important

When performing searches, *string* allows *regular expression* syntax. While simple text (such as passwd) works as expected, regular expressions use meta-characters (such as $, \*, ., and ^) for more sophisticated pattern matching. Therefore, searching with strings that include program expression meta-characters, such as make $$$, might yield unexpected results.

Regular expressions and syntax are discussed in *Red Hat System Administration II*, and in the regex(7) man topic.

**Reading Man Pages**

Each topic is separated into several parts. Most topics share the same headings and are presented in the same order. Typically a topic does not feature all headings, because not all headings apply for all topics.

Common headings are:

**Table 4.3. Headings**

| **Heading** | **Description** |
| --- | --- |
| NAME | Subject name. Usually a command or file name. Very brief description. |
| SYNOPSIS | Summary of the command syntax. |
| DESCRIPTION | In-depth description to provide a basic understanding of the topic. |
| OPTIONS | Explanation of the command execution options. |
| EXAMPLES | Examples of how to use the command, function, or file. |
| FILES | A list of files and directories related to the man page. |
| SEE ALSO | Related information, normally other man page topics. |
| BUGS | Known bugs in the software. |
| AUTHOR | Information about who has contributed to the development of the topic. |

Searching for man pages by keyword

A keyword search of man pages is performed with **man -k *keyword***, which displays a list of keyword-matching man page topics with section numbers.

**[student@desktopX ~]$** **man -k passwd**

checkPasswdAccess (3) - query the SELinux policy database in the kernel.

chpasswd (8) - update passwords in batch mode

ckpasswd (8) - nnrpd password authenticator

fgetpwent\_r (3) - get passwd file entry reentrantly

getpwent\_r (3) - get passwd file entry reentrantly

...

passwd (1) - update user's authentication tokens

sslpasswd (1ssl) - compute password hashes

passwd (5) - password file

passwd.nntp (5) - Passwords for connecting to remote NNTP servers

passwd2des (3) - RFS password encryption

...

Popular system administration topics are in sections 1 (user commands), 5 (file formats), and 8 (administrative commands). Administrators using certain troubleshooting tools also use section 2 (system calls). The remaining sections are generally for programmer reference or advanced administration.

Note

Keyword searches rely on an index generated by the mandb(8) command, which must be run as root. The command runs daily through cron.daily, or by **anacrontab** within an hour of boot, if out of date.

Important

The **man** command -K (uppercase) option performs a full-text page search, not just titles and descriptions like the -k option. A full-text search uses greater system resources and take more time.

References

## man(1), mandb(8), mGuided Exercise: Reading Manual Pages

In this exercise, you will practice finding relevant information by using **man** options and arguments.

**Outcomes**

You should be able to use the **man** Linux manual system and find useful information by searching and browsing.

Log in to workstation as student using student as the password.

On workstation, run the **lab help-manual start** command. It creates a file called manual.

**[student@workstation ~]$ lab help-manual start**

1. On workstation view the **gedit** man page. View the options for editing a specific file using **gedit** from the command line.

Use one of the options from the **gedit** man page to open the /home/student/manual file using **gedit** with the cursor at the end of the file.

* 1. View the **gedit** man page.

**[student@workstation ~]$ man gedit**

GEDIT(1) General Commands Manual GEDIT(1)

NAME

gedit - text editor for the GNOME Desktop

SYNOPSIS

gedit [OPTION...] [FILE...] [+LINE[:COLUMN]]

gedit [OPTION...] -

*...output omitted...*

* 1. In the **gedit** man page, learn the options for editing a specific file from the command line.
  2. *...output omitted...*
  3. **FILE** Specifies the file to open when gedit starts.
  4. *...output omitted...*
  5. **+LINE** For the first file, go to the line specified by LINE (do not insert a space between the "+" sign and the number). If LINE is missing, go to the last line.

*...output omitted...*

Press **q** to quit the man page.

* 1. Use the **gedit +** command to open the manual file. The missing line number next to *+* option opens a file passed as an argument with cursor at the end of the last line.

**[student@workstation ~]$ gedit + manual**

the quick brown fox just came over to greet the lazy poodle!

Confirm that the file is opened with the cursor at the end of the last line in the file. Press **Ctrl**+**q** to close the application.

1. Read the su(1) man page.

Note that when the *user* is omitted the **su** command assumes the user is root. If the **su** command is followed by a single dash (-), it starts a child login shell. Without the dash, a non-login child shell is created that matches the user's current environment.

**[student@workstation ~]$ man 1 su**

SU(1) User Commands SU(1)

NAME

su - run a command with substitute user and group ID

SYNOPSIS

su [options] [-] [user [argument...]]

DESCRIPTION

su allows to run commands with a substitute user and group ID.

When called without arguments, su defaults to running an interactive

shell as root.

*...output omitted...*

OPTIONS

*...output omitted...*

**-, -l, --login**

Start the shell as a login shell with an environment similar to a real login

*...output omitted...*

Note

Note that comma-separated options on a single line, such as -, -l, and --login, all result in the same behavior.

Press **q** to quit the man page.

1. The **man** command also has its own manual pages.
2. **[student@workstation ~]$ man man**
3. MAN(1) Manual pager utils MAN(1)
4. NAME
5. man - an interface to the on-line reference manuals
6. *...output omitted...*
7. DESCRIPTION
8. man is the system's manual pager. Each page argument given to man is
9. normally the name of a program, utility or function. The manual page
10. associated with each of these arguments is then found and displayed.
11. A section, if provided, will direct man to look only in that section
12. of the manual.

*...output omitted...*

Press **q** to quit the man page.

1. All man pages are located in /usr/share/man. Locate the binary, source, and manual pages located in the /usr/share/man directory by using the **whereis** command.
2. **[student@workstation ~]$ whereis passwd**

passwd: /usr/bin/passwd /etc/passwd /usr/share/man/man1/passwd.1.gz /usr/share/man/man5/passwd.5.gz

1. Use the **man -k zip** command to list detailed information about a ZIP archive.
2. **[student@workstation ~]$ man -k zip**
3. *...output omitted...*
4. **zipinfo (1) - list detailed information about a ZIP archive**
5. zipnote (1) - write the comments in zipfile to stdout, edit comments and rename files in zipfile

zipsplit (1) - split a zipfile into smaller zipfiles

1. Use the **man -k boot** to list the man page containing a list of parameters that can be passed to the kernel at boot time.
2. **[student@workstation ~]$ man -k boot**
3. *...output omitted...*
4. bootctl (1) - Control the firmware and boot manager settings
5. **bootparam (7) - introduction to boot time parameters of the Linux kernel**
6. bootup (7) - System bootup process

*...output omitted...*

1. Use the **man -k ext4** to find the command used to tune ext4 file system parameters.
2. **[student@workstation ~]$ man -k ext4**
3. *...output omitted...*
4. resize2fs (8) - ext2/ext3/ext4 file system resizer

**tune2fs (8) - adjust tunable filesystem parameters on ext2/ext3/ext4 filesystems**

**Finish**

On workstation, run the **lab help-manual finish** script to complete this exercise.

**[student@workstation ~]$ lab help-manual finish**

This concludes the guided exercise.

an-pages(7), less(1), intro(1), intro(2), intro(5), intro(7), intro(8) man pages

Reading Info Documentation

Objectives

After completing this section, students should be able to find information from local documentation in GNU Info.

Introducing GNU Info

Man pages have a format useful as a command reference, but less useful as general documentation. For such documents, the GNU Project developed a different online documentation system, known as *GNU Info*. Info documents are an important resource on a Red Hat Enterprise Linux system because many fundamental components and utilities, such as the coreutils package and glibc standard libraries, are either developed by the GNU Project or utilize the Info document system.

Important

You might wonder why there are two local documentation systems, man pages and Info documents. Some of the reasons for this are practical in nature, and some have to do with the way Linux and its applications have been developed by various open source communities over the years.

Man pages have a much more formal format, and typically document a specific command or function from a software package, and are structured as individual text files. Info documents typically cover particular software packages as a whole, tend to have more practical examples of how to use the software, and are structured as hypertext documents.

You should be familiar with both systems in order to take maximum advantage of the information available to you from the system.

**Reading Info Documentation**

To launch the Info document viewer, use the **pinfo** command. **pinfo** opens in the *top directory*.

|  |
| --- |
|  |

Figure 4.1: pinfo Info document viewer, top directory

Info documentation is comprehensive and hyperlinked. It is possible to output info pages to multiple formats. By contrast, man pages are optimized for printed output. The Info format is more flexible than man pages, allowing thorough discussion of complex commands and concepts. Like man pages, Info nodes are read from the command line, using the **pinfo** command.

A typical man page has a small amount of content focusing on one particular topic, command, tool, or file. The Info documentation is a comprehensive document. Info provides the following improvements:

* One single document for a large system containing all the necessary information for that system
* Hyperlinks
* A complete browsable document index
* A full text search of the entire document

Some commands and utilities have both **man** pages and info documentation; usually, the Info documentation is more in depth. Compare the differences in **tar** documentation using **man** and **pinfo**:

**[user@host ~]$** **man tar**

**[user@host ~]$** **pinfo tar**

The **pinfo** reader is more advanced than the original **info** command. To browse a specific topic, use the **pinfo *topic*** command. The **pinfo** command without an argument opens the top directory. New documentation becomes available in **pinfo** when their software packages are installed.

Note

If no Info topic exists in the system for a particular entry that you requested, Info will look for a matching man page and display that instead.

Comparing GNU Info and Man Page Navigation

The **pinfo** command and the **man** command use slightly different navigational keystrokes. The following table compares the navigational keystrokes for both commands:

**Table 4.4. pinfo and man, key binding comparison**

| **Navigation** | **pinfo** | **man** |
| --- | --- | --- |
| Scroll forward (down) one screen | **PageDown** or **Space** | **PageDown** or **Space** |
| Scroll backward (up) one screen | **PageUp** or **b** | **PageUp** or **b** |
| Display the directory of topics | **d** | - |
| Scroll forward (down) one half-screen | - | **d** |
| Display the parent node of a topic | **u** | - |
| Display the top (up) of a topic | **HOME** | **g** |
| Scroll backward (up) one half-screen | - | **u** |
| Scroll forward (down) to next hyperlink | **DownArrow** | - |
| Open topic at cursor location | **Enter** | - |
| Scroll forward (down) one line | - | **DownArrow** or **Enter** |
| Scroll backward (up) to previous hyperlink | **UpArrow** | - |
| Scroll backward (up) one line | - | **UpArrow** |
| Search for a pattern | **/***string* | **/***string* |
| Display next node (chapter) in topic | **n** | - |
| Repeat previous search forward (down) | **/** then **Enter** | **n** |
| Display previous node (chapter) in topic | **p** | - |
| Repeat previous search backward (up) | - | **N** |
| Quit the program | **q** | **q** |

References

**pinfo info** (*Info: An Introduction*)

**pinfo pinfo** (*Documentation for****pinfo***)

The GNU Project <http://www.gnu.org/gnu/thegnuproject.html>

pinfo(1) and info(1) man pages

Guided Exercise: Reading Info Documentation

In this exercise, you will look up information stored in GNU Info documents by navigating those documents with command-line tools.

**Outcomes**

You should be able to navigate GNU Info documentation with command-line tools.

Log in to workstation as student using student as the password.

On workstation, run the **lab help-info start** command.

**[student@workstation ~]$ lab help-info start**

1. On workstation launch **pinfo** without any arguments.

**[student@workstation ~]$ pinfo**

1. Navigate to the Common options topic.

Use **UpArrow** or **DownArrow** until (coreutils) Common options is highlighted.

|  |
| --- |
|  |

Figure 4.2: Bash documentation

1. Press **Enter** to view this topic.

|  |
| --- |
|  |

1. Figure 4.3: Common options info topic
2. Browse through this Info topic. Learn whether long-style options can be abbreviated.

Use **PageUp** and **PageDown** to navigate through the topic. Yes, many programs allow long options to be abbreviated.

1. Determine what the symbols -- signify when used as a command argument.

The symbols -- signify the end of command *options* and the start of command *arguments* in complex commands where the shell's command-line parser might not correctly make the distinction.

1. Without exiting **pinfo**, move up to the GNU Coreutils node.

Press **u** to move up to the top node of the topic.

1. Return to the top level topic.

Press **u** again. Observe that when positioned at the top of a topic node, moving up returns to the directory of topics. Alternately, pressing **d** from any level or topic moves directly to the directory of topics.

1. Search for the pattern coreutils and select that topic.

Press **/** followed by the search pattern “coreutils”. With the topic highlighted, press **Enter**.

|  |
| --- |
|  |

Figure 4.4: Search result

1. In the menu at the top, locate and select Output of entire files by pressing **n**. Browse the topic.

Use **Enter** to select cat invocation. Use the arrow keys to browse the topic.

1. Move up two levels to return to GNU Coreutils. Move to Summarizing files.

Press **Enter** to select the topic then browse the topic.

1. Press **q** to quit **pinfo**.
2. Use the **pinfo** command again, specifying coreutils as the destination topic from the command line.

**[student@workstation ~]$ pinfo coreutils**

1. Select the Disk usage topic.

Press **DownArrow** to highlight Disk usage, then press **Enter** to select this topic.

1. Read the df invocation and du invocation subtopics.

Use arrow keys to highlight a topic, **PageUp** and **PageDown** to browse the text, then press **u** to move up one level. Press **q** to quit when you are finished.

**Finish**

On workstation, run the **lab help-info finish** script to complete this exercise.

**[student@workstation ~]$ lab help-info finish**

This concludes the guided exercise.

Summary

In this chapter, you learned:

* Man pages are viewed with the **man** command and provide information on components of a Linux system, such as files, commands, and functions.
* By convention, when referring to a man page the name of a page is followed by its section number in parentheses.
* Info documents are viewed with the **pinfo** command and are made up of a collection of hypertext nodes, providing information about software packages as a whole.
* The navigational keystrokes used by **man** and **pinfo** are slightly different.

Chapter 5. Creating, Viewing, and Editing Text Files

[**Redirecting Output to a File or Program**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch05/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Quiz: Redirecting Output to a File or Program**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch05s02/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Editing Text Files from the Shell Prompt**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch05s03/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Guided Exercise: Editing Text Files from the Shell Prompt**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch05s04/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Changing the Shell Environment**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch05s05/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Guided Exercise: Changing the Shell Environment**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch05s06/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Lab: Creating, Viewing, and Editing Text Files**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch05s07/9a2ef70f-4e72-42df-a498-b694b274af27)

[**Summary**](https://rha.ole.redhat.com/rha/app/courses/rh124-8.2/pages/ch05s08/9a2ef70f-4e72-42df-a498-b694b274af27)

**Abstract**

|  |  |
| --- | --- |
| **Goal** | Create, view, and edit text files from command output or in a text editor. |
| **Objectives** | * Save command output or errors to a file with shell redirection, and process command output through multiple command-line programs with pipes. * Create and edit text files using the **vim** editor. * Use shell variables to help run commands, and edit Bash startup scripts to set shell and environment variables to modify the behavior of the shell and programs run from the shell. |
| **Sections** | * Redirecting Output to a File or Program (and Quiz) * Editing Text Files from the Shell Prompt (and Guided Exercise) * Changing the Shell Environment (and Guided Exercise) |
| **Lab** | Creating, Viewing, and Editing Text Files |

Redirecting Output to a File or Program

Objectives

After completing this section, you should be able to save output or errors to a file with shell redirection, and process command output through multiple command-line programs with pipes.

Standard Input, Standard Output, and Standard Error

A running program, or *process*, needs to read input from somewhere and write output to somewhere. A command run from the shell prompt normally reads its input from the keyboard and sends its output to its terminal window.

A process uses numbered channels called *file descriptors* to get input and send output. All processes start with at least three file descriptors. *Standard input* (channel 0) reads input from the keyboard. *Standard output* (channel 1) sends normal output to the terminal. *Standard error* (channel 2) sends error messages to the terminal. If a program opens separate connections to other files, it may use higher-numbered file descriptors.

|  |
| --- |
|  |

Figure 5.1: Process I/O channels (file descriptors)

**Table 5.1. Channels (File Descriptors)**

| **Number** | **Channel name** | **Description** | **Default connection** | **Usage** |
| --- | --- | --- | --- | --- |
| 0 | stdin | Standard input | Keyboard | read only |
| 1 | stdout | Standard output | Terminal | write only |
| 2 | stderr | Standard error | Terminal | write only |
| 3+ | *filename* | Other files | none | read and/or write |

Redirecting Output to a File

*I/O redirection* changes how the process gets its input or output. Instead of getting input from the keyboard, or sending output and errors to the terminal, the process reads from or writes to files. Redirection lets you save messages to a file that are normally sent to the terminal window. Alternatively, you can use redirection to discard output or errors, so they are not displayed on the terminal or saved.

Redirecting stdout suppresses process output from appearing on the terminal. As seen in the following table, redirecting *only* stdout does not suppress stderr error messages from displaying on the terminal. If the file does not exist, it will be created. If the file does exist and the redirection is not one that appends to the file, the file's contents will be overwritten.

If you want to discard messages, the special file /dev/null quietly discards channel output redirected to it and is always an empty file.

**Table 5.2. Output Redirection Operators**

| **Usage** | **Explanation** | **Visual aid** |
| --- | --- | --- |
| > *file* | redirect stdout to overwrite a file | |  | | --- | |  | |
| >> *file* | redirect stdout to append to a file | |  | | --- | |  | |
| 2> *file* | redirect stderr to overwrite a file | |  | | --- | |  | |
| 2> /dev/null | discard stderr error messages by redirecting to /dev/null | |  | | --- | |  | |
| > *file*  2>&1 | redirect stdout and stderr to overwrite the same file | |  | | --- | |  | |
| &> *file* |
| >> *file*  2>&1 | redirect stdout and stderr to append to the same file | |  | | --- | |  | |
| &>> *file* |

**Important**

The order of redirection operations is important. The following sequence redirects standard output to file and then redirects standard error to the same place as standard output (file).

> file 2>&1

However, the next sequence does redirection in the opposite order. This redirects standard error to the default place for standard output (the terminal window, so no change) and *then* redirects only standard output to file.

2>&1 > file

Because of this, some people prefer to use the merging redirection operators:

|  |  |  |
| --- | --- | --- |
| &>file | instead of | >file 2>&1 |
| &>>file | instead of | >>file 2>&1 (in Bash 4 / RHEL 6 and later) |

However, other system administrators and programmers who also use other shells related to **bash** (known as Bourne-compatible shells) for scripting commands think that the newer merging redirection operators should be avoided, because they are not standardized or implemented in all of those shells and have other limitations.

The authors of this course take a neutral stance on this topic, and both syntaxes are likely to be encountered in the field.

**Examples for Output Redirection**

Many routine administration tasks are simplified by using redirection. Use the previous table to assist while considering the following examples:

* Save a time stamp for later reference.

**[user@host ~]$ date > /tmp/saved-timestamp**

* Copy the last 100 lines from a log file to another file.

**[user@host ~]$ tail -n 100 /var/log/dmesg > /tmp/last-100-boot-messages**

* Concatenate four files into one.

**[user@host ~]$ cat file1 file2 file3 file4 > /tmp/all-four-in-one**

* List the home directory's hidden and regular file names into a file.

**[user@host ~]$ ls -a > /tmp/my-file-names**

* Append output to an existing file.

**[user@host ~]$ echo "new line of information" >> /tmp/many-lines-of-information**

**[user@host ~]$ diff previous-file current-file >> /tmp/tracking-changes-made**

* The next few commands generate error messages because some system directories are inaccessible to normal users. Observe as the error messages are redirected. Redirect errors to a file while viewing normal command output on the terminal.

**[user@host ~]$ find /etc -name passwd 2> /tmp/errors**

* Save process output and error messages to separate files.

**[user@host ~]$ find /etc -name passwd > /tmp/output 2> /tmp/errors**

* Ignore and discard error messages.

**[user@host ~]$ find /etc -name passwd > /tmp/output 2> /dev/null**

* Store output and generated errors together.

**[user@host ~]$ find /etc -name passwd &> /tmp/save-both**

* Append output and generated errors to an existing file.

**[user@host ~]$ find /etc -name passwd >> /tmp/save-both 2>&1**

Constructing Pipelines

A *pipeline* is a sequence of one or more commands separated by the *pipe* character (|). A pipe connects the standard output of the first command to the standard input of the next command.

|  |
| --- |
|  |

Figure 5.8: Process I/O piping

Pipelines allow the output of a process to be manipulated and formatted by other processes before it is output to the terminal. One useful mental image is to imagine that data is "flowing" through the pipeline from one process to another, being altered slightly by each command in the pipeline through which it flows.

Note

Pipelines and I/O redirection both manipulate standard output and standard input. Redirection sends standard output to files or gets standard input from files. Pipes send the standard output from one process to the standard input of another process.

**Pipeline Examples**

This example takes the output of the **ls** command and uses **less** to display it on the terminal one screen at a time.

**[user@host ~]$ ls -l /usr/bin | less**

The output of the **ls** command is piped to **wc -l**, which counts the number of lines received from **ls** and prints that to the terminal.

**[user@host ~]$ ls | wc -l**

In this pipeline, **head** will output the first 10 lines of output from **ls -t**, with the final result redirected to a file.

**[user@host ~]$ ls -t | head -n 10 > /tmp/ten-last-changed-files**

**Pipelines, Redirection, and the tee Command**

When redirection is combined with a pipeline, the shell sets up the entire pipeline first, then it redirects input/output. If output redirection is used in the *middle* of a pipeline, the output will go to the file and not to the next command in the pipeline.

In this example, the output of the **ls** command goes to the file, and **less** displays nothing on the terminal.

**[user@host ~]$ ls > /tmp/saved-output | less**

The **tee** command overcomes this limitation. In a pipeline, **tee** copies its standard input to its standard output and also redirects its standard output to the files named as arguments to the command. If you imagine data as water flowing through a pipeline, **tee** can be visualized as a "T" joint in the pipe which directs output in two directions.

|  |
| --- |
|  |

Figure 5.9: Process I/O piping with tee

**Pipeline Examples Using the tee Command**

This example redirects the output of the **ls** command to the file and passes it to **less** to be displayed on the terminal one screen at a time.

**[user@host ~]$ ls -l | tee /tmp/saved-output | less**

If **tee** is used at the end of a pipeline, then the final output of a command can be saved and output to the terminal at the same time.

**[user@host ~]$ ls -t | head -n 10 | tee /tmp/ten-last-changed-files**

Important

Standard error can be redirected through a pipe, but the merging redirection operators (&> and &>>) cannot be used to do this.

The following is the correct way to redirect both standard output and standard error through a pipe:

**[user@host ~]$** find -name / passwd 2>&1 | less

References

**info bash** (*The GNU Bash Reference Manual*)

* Section 3.2.2: Pipelines
* Section 3.6: Redirections

**info coreutils 'tee invocation'** (*The GNU coreutils Manual*)

* Section 17.1: Redirect output to multiple files or processes

**bash**(1), **cat**(1), **head**(1), **less**(1), **mail**(1), **tee**(1), **tty**(1), **wc**(1) man pages

Quiz: Redirecting Output to a File or Program

Choose the correct answer to the following questions:

1. Which answer displays output to a terminal and ignores all errors?

A &>file

B 2> &>file

C 2>/dev/null

D 1>/dev/null

2. Which answer sends output to a file and sends errors to a different file?

A >file 2>file2

B >file 1>file2

C >file &2>file2

D | tee file

3. Which answer sends both output and errors to a file, creating it or overwriting its contents?

A | tee file

B 2 &>file

C 1 &>file

D &>file

4. Which answer sends output and errors to the same file ensuring existing file content is preserved?

A >file 2>file2

B &>file

C >>file 2>&1

D >>file 1>&1

5. Which answer discards all messages normally sent to the terminal?

A >file 2>file2

B &>/dev/null

C &>/dev/null 2>file

D &>file

6. Which answer sends output to both the screen and a file at the same time?

A &>/dev/null

B >file 2>file2

C | tee file

D | < file

7. Which answer saves output to a file and discards error messages?

A &>file

B | tee file 2> /dev/null

C > file 1> /dev/null

D > file 2> /dev/null

**Editing Text Files from the Shell Prompt**

Objectives

After completing this section, you should be able to create and edit text files from the command line using the **vim** editor.

Editing Files with Vim

A key design principle of Linux is that information and configuration settings are commonly stored in text-based files. These files can be structured in various ways, as lists of settings, in INI-like formats, as structured XML or YAML, and so on. However, the advantage of text files is that they can be viewed and edited using any simple text editor.

Vim is an improved version of the **vi** editor distributed with Linux and UNIX systems. Vim is highly configurable and efficient for practiced users, including such features as split screen editing, color formatting, and highlighting for editing text.

**Why Learn Vim?**

You should know how to use at least one text editor that can be used from a text-only shell prompt. If you do, you can edit text-based configuration files from a terminal window, or from remote logins through **ssh** or the Web Console. Then you do not need access to a graphical desktop in order to edit files on a server, and in fact that server might not need to run a graphical desktop environment at all.

But then, why learn Vim instead of other possible options? The key reason is that Vim is almost always installed on a server, if any text editor is present. This is because **vi** was specified by the POSIX standard that Linux and many other UNIX-like operating systems comply with in large part.

In addition, Vim is often used as the **vi** implementation on other common operating systems or distributions. For example, macOS currently includes a lightweight installation of Vim by default. So Vim skills learned for Linux might also help you get things done elsewhere.

**Starting Vim**

Vim may be installed in Red Hat Enterprise Linux in two different ways. This can affect the features and Vim commands available to you.

Your server might only have the vim-minimal package installed. This is a very lightweight installation that includes only the core feature set and the basic **vi** command. In this case, you can open a file for editing with **vi *filename***, and all the core features discussed in this section will be available to you.

Alternatively, your server might have the vim-enhanced package installed. This provides a much more comprehensive set of features, an on-line help system, and a tutorial program. In order to start Vim in this enhanced mode, you use the **vim** command.

**[user@host ~]$ vim *filename***

Either way, the core features that we will discuss in this section will work with both commands.

Note

If vim-enhanced is installed, regular users will have a shell alias set so that if they run the **vi** command, they will automatically get the **vim** command instead. This does not apply to root and other users with UIDs below 200 (which are used by system services).

If you are editing files as the root user and you expect **vi** to run in enhanced mode, this can be a surprise. Likewise, if vim-enhanced is installed and a regular user wants the simple **vi** for some reason, they might need to use **\vi** to override the alias temporarily.

Advanced users can use **\vi --version** and **vim --version** to compare the feature sets of the two commands.

**Vim Operating Modes**

An unusual characteristic of Vim is that it has several *modes* of operation, including *command mode*, *extended command mode*, *edit mode*, and *visual mode*. Depending on the mode, you may be issuing commands, editing text, or working with blocks of text. As a new Vim user, you should always be aware of your current mode as keystrokes have different effects in different modes.

|  |
| --- |
|  |

Figure 5.10: Moving between Vim modes

When you first open Vim, it starts in *command mode*, which is used for navigation, cut and paste, and other text manipulation. Enter each of the other modes with single character keystrokes to access specific editing functionality:

* An **i** keystroke enters *insert mode*, where all text typed becomes file content. Pressing **Esc** returns to command mode.
* A **v** keystroke enters *visual mode*, where multiple characters may be selected for text manipulation. Use **Shift**+**V** for multiline and **Ctrl**+**V** for block selection. The same keystroke used to enter visual mode (**v**, **Shift**+**V** or **Ctrl**+**V**) is used to exit.
* The **:** keystroke begins *extended command mode* for tasks such as writing the file (to save it), and quitting the Vim editor.

Note

If you are not sure what mode Vim is in, you can try pressing **Esc** a few times to get back into command mode. Pressing **Esc** in command mode is harmless, so a few extra key presses are okay.

**The Minimum, Basic Vim Workflow**

Vim has efficient, coordinated keystrokes for advanced editing tasks. Although considered useful with practice, Vim's capabilities can overwhelm new users.

The **i** key puts Vim into insert mode. All text entered after this is treated as file contents until you exit insert mode. The **Esc** key exits insert mode and returns Vim to command mode. The **u** key will undo the most recent edit. Press the **x** key to delete a single character. The **:w** command writes (saves) the file and remains in command mode for more editing. The **:wq** command writes (saves) the file and quits Vim. The **:q!** command quits Vim, discarding all file changes since the last write. The Vim user must learn these commands to accomplish any editing task.

**Rearranging Existing Text**

In Vim, copy and paste is known as *yank and put*, using command characters **y** and **p**. Begin by positioning the cursor on the first character to be selected, and then enter visual mode. Use the arrow keys to expand the visual selection. When ready, press **y** to *yank* the selection into memory. Position the cursor at the new location, and then press **p** to *put* the selection at the cursor.

**Visual Mode in Vim**

Visual mode is a great way to highlight and manipulate text. There are three keystrokes:

* Character mode: **v**
* Line mode: **Shift**+**v**
* Block mode: **Ctrl**+**v**

Character mode highlights sentences in a block of text. The word VISUAL will appear at the bottom of the screen. Press **v** to enter visual character mode. **Shift**+**v** enters line mode. VISUAL LINE will appear at the bottom of the screen.

Visual block mode is perfect for manipulating data files. From the cursor, press the **Ctrl**+**v** to enter visual block. VISUAL BLOCK will appear at the bottom of the screen. Use the arrow keys to highlight the section to change.

Note

Vim has a lot of capabilities, but you should master the basic workflow first. You do not need to quickly understand the entire editor and its capabilities. Get comfortable with those basics through practice and then you can expand your Vim vocabulary by learning additional Vim commands (keystrokes).

The exercise for this section will introduce you to the **vimtutor** command. This tutorial, which ships with vim-enhanced, is an excellent way to learn the core functionality of Vim.

References

**vim**(1) man page

The **:help** command in **vim** (if the vim-enhanced package is installed).

[Vim the editor](http://www.vim.org/)

[Getting Started with Vim visual mode](https://opensource.com/article/19/2/getting-started-vim-visual-mode)

Guided Exercise: Editing Text Files from the Shell Prompt

In this exercise, you will use **vimtutor** to practice basic editing techniques in the vim editor.

**Outcomes**

You should be able to:

* Edit files using Vim.
* Gain competency in Vim using **vimtutor**.

Log in to workstation as student using student as the password.

On workstation, run the **lab edit-vim start** command. This script verifies that the target server is running.

**[student@workstation ~]$ lab edit-vim start**

1. Use the **ssh** command to log in to servera.
2. **[student@workstation ~]$ ssh student@servera**
3. *...output omitted...*

[student@servera ~]$

1. Open **vimtutor**. Read the Welcome screen and perform *Lesson 1.1*.

**[student@servera ~]$ vimtutor**

In the presentation, keyboard arrow keys are used for navigation. When **vi** was first developed, users could not rely on having arrow keys or working keyboard mappings for arrow keys to move the cursor. Therefore, **vi** was originally designed to move the cursor using commands using standard character keys, such as the conveniently grouped **H**, **J**, **K**, and **L**.

Here is one way to remember them:

hang *back*, jump *down*, kick *up*, leap *forward*.

1. In the **vimtutor** window, perform *Lesson 1.2*.

This lesson teaches users how to quit without keeping unwanted changes. All changes are lost. Sometimes this is preferable to leaving a critical file in an incorrect state.

1. In the **vimtutor** window, perform *Lesson 1.3*.

Vim has fast, efficient keystrokes to delete an exact amount of words, lines, sentences, and paragraphs. However, any editing job *can* be accomplished using x for single character deletion.

1. In the **vimtutor** window, perform *Lesson 1.4*.

For most editing tasks, the first key pressed is i.

1. In the **vimtutor** window, perform *Lesson 1.5*.

In the lecture, only the **i** (*insert*) command was taught as the keystroke to enter edit mode. This **vimtutor** lesson demonstrates other available keystrokes to change the cursor placement when insert mode is entered. In insert mode, all typed text is file content.

1. In the **vimtutor** window, perform *Lesson 1.6*.

Type **:wq** to save the file and quit the editor.

1. In the **vimtutor** window, read the *Lesson 1 Summary*.

The **vimtutor** command includes six more multistep lessons. These lessons are not assigned as part of this course but feel free to explore them on your own to learn more.

1. Exit from servera.
2. **[student@servera ~]$ exit**
3. logout
4. Connection to servera closed.

[student@workstation ~]$

**Finish**

On workstation, run the **lab edit-vim finish** script to complete this exercise.

**[student@workstation ~]$ lab edit-vim finish**

This concludes the guided exercise.

Changing the Shell Environment

Objectives

After completing this section, you should be able to set shell variables to help run commands, and edit Bash startup scripts to set shell and environment variables to modify the behavior of the shell and programs run from the shell.

Using Shell Variables

The Bash shell allows you to set *shell variables* that you can use to help run commands or to modify the behavior of the shell. You can also export shell variables as *environment variables*, which are automatically copied to programs run from that shell when they start. You can use variables to help make it easier to run a command with a long argument, or to apply a common setting to commands run from that shell.

Shell variables are unique to a particular shell session. If you have two terminal windows open, or two independent login sessions to the same remote server, you are running two shells. Each shell has its own set of values for its shell variables.

**Assigning Values to Variables**

Assign a value to a shell variable using the following syntax:

VARIABLENAME=value

Variable names can contain uppercase or lowercase letters, digits, and the underscore character (\_). For example, the following commands set shell variables:

**[user@host ~]$ COUNT=40**

**[user@host ~]$ first\_name=John**

**[user@host ~]$ file1=/tmp/abc**

**[user@host ~]$ \_ID=RH123**

Remember, this change only affects the shell in which you run the command, not any other shells you may be running on that server.

You can use the **set** command to list all shell variables that are currently set. (It also lists all shell functions, which you can ignore.) This list is long enough that you may want to pipe the output into the **less** command so that you can view it one page at a time.

**[user@host ~]$ set | less**

BASH=/usr/bin/bash

BASHOPTS=checkwinsize:​cmdhist:​complete\_fullquote:​expand\_aliases:​extglob:​extquote:​force\_fignore:​histappend:​interactive\_comments:​progcomp:​promptvars:​sourcepath

BASHRCSOURCED=Y

*...output omitted...*

**Retrieving Values with Variable Expansion**

You can use *variable expansion* to refer to the value of a variable that you have set. To do this, precede the name of the variable with a dollar sign ($). In the following example, the **echo** command prints out the rest of the command line entered, but after variable expansion is performed.

For example, the following command sets the variable COUNT to 40.

**[user@host ~]$ COUNT=40**

If you enter the command **echo COUNT**, it will print out the string COUNT.

**[user@host ~]$ echo COUNT**

COUNT

But if you enter the command **echo $COUNT**, it will print out the value of the variable COUNT.

**[user@host ~]$ echo $COUNT**

40

A more practical example might be to use a variable to refer to a long file name for multiple commands.

**[user@host ~]$ file1=/tmp/tmp.z9pXW0HqcC**

**[user@host ~]$ ls -l $file1**

-rw-------. 1 student student 1452 Jan 22 14:39 /tmp/tmp.z9pXW0HqcC

**[user@host ~]$ rm $file1**

**[user@host ~]$ ls -l $file1**

total 0

Important

If there are any trailing characters adjacent to the variable name, you might need to protect the variable name with curly braces. You can always use curly braces in variable expansion, but you will also see many examples in which they are not needed and are omitted.

In the following example, the first **echo** command tries to expand the nonexistent variable COUNTx, which does not cause an error but instead returns nothing.

**[user@host ~]$ echo Repeat $COUNTx**

Repeat

**[user@host ~]$ echo Repeat ${COUNT}x**

Repeat 40x

**Configuring Bash with Shell Variables**

Some shell variables are set when Bash starts but can be modified to adjust the shell's behavior.

For example, two shell variables that affect the shell history and the **history** command are HISTFILE and HISTFILESIZE. If HISTFILE is set, it specifies the location of a file to save the shell history in when it exits. By default this is the user's ~/.bash\_history file. The HISTFILESIZE variable specifies how many commands should be saved in that file from the history.

Another example is PS1, which is a shell variable that controls the appearance of the shell prompt. If you change this value, it will change the appearance of your shell prompt. A number of special character expansions supported by the prompt are listed in the "PROMPTING" section of the **bash**(1) man page.

**[user@host ~]$ PS1="bash\$ "**

**bash$ PS1="[\u@\h \W]\$ "**

**[user@host ~]$**

Two items to note about the above example: first, because the value set by PS1 is a prompt, it is virtually always desirable to end the prompt with a trailing space. Second, whenever the value of a variable contains some form of space, including a space, a tab, or a return, the value must be surrounded by quotes, either single or double; this is not optional. Unexpected results will occur if the quotes are omitted. Examine the PS1 example above and note that it conforms to both the recommendation (trailing space) and the rule (quotes).

Configuring Programs with Environment Variables

The shell provides an *environment* to the programs you run from that shell. Among other things, this environment includes information on the current working directory on the file system, the command-line options passed to the program, and the values of *environment variables*. The programs may use these environment variables to change their behavior or their default settings.

Shell variables that are not environment variables can only be used by the shell. Environment variables can be used by the shell *and* by programs run from that shell.

Note

HISTFILE, HISTFILESIZE, and PS1, learned in the previous section, do not need to be exported as environment variables because they are only used by the shell itself, not by the programs that you run from the shell.

You can make any variable defined in the shell into an environment variable by marking it for export with the **export** command.

**[user@host ~]$ EDITOR=vim**

**[user@host ~]$ export EDITOR**

You can set and export a variable in one step:

**[user@host ~]$ export EDITOR=vim**

Applications and sessions use these variables to determine their behavior. For example, the shell automatically sets the HOME variable to the file name of the user's home directory when it starts. This can be used to help programs determine where to save files.

Another example is LANG, which sets the locale. This adjusts the preferred language for program output; the character set; the formatting of dates, numbers, and currency; and the sort order for programs. If it is set to en\_US.UTF-8, the locale will use US English with UTF-8 Unicode character encoding. If it is set to something else, for example fr\_FR.UTF-8, it will use French UTF-8 Unicode encoding.

**[user@host ~]$ date**

Tue Jan 22 16:37:45 CST 2019

**[user@host ~]$ export LANG=fr\_FR.UTF-8**

**[user@host ~]$ date**

mar. janv. 22 16:38:14 CST 2019

Another important environment variable is PATH. The PATH variable contains a list of colon-separated directories that contain programs:

**[user@host ~]$ echo $PATH**

/home/user/.local/bin:/home/user/bin:/usr/share/Modules/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin

When you run a command such as **ls**, the shell looks for the executable file ls in each of those directories in order, and runs the first matching file it finds. (On a typical system, this is /usr/bin/ls.)

You can easily add additional directories to the end of your PATH. For example, perhaps you have executable programs or scripts that you want to run like regular commands in /home/user/sbin. You can add /home/user/sbin to the end of your PATH for the current session like this:

**[user@host ~]$ export PATH=${PATH}:/home/user/sbin**

To list all the environment variables for a particular shell, run the **env** command:

**[user@host ~]$ env**

*...output omitted...*

LANG=en\_US.UTF-8

HISTCONTROL=ignoredups

HOSTNAME=host.example.com

XDG\_SESSION\_ID=4

*...output omitted...*

**Setting the Default Text Editor**

The EDITOR environment variable specifies the program you want to use as your default text editor for command-line programs. Many programs use **vi** or **vim** if it is not specified, but you can override this preference if required:

**[user@host ~]$ export EDITOR=nano**

Important

By convention, environment variables and shell variables that are automatically set by the shell have names that use all uppercase characters. If you are setting your own variables, you may want to use names made up of lowercase characters to help avoid naming collisions.

Setting Variables Automatically

If you want to set shell or environment variables automatically when your shell starts, you can edit the Bash startup scripts. When Bash starts, several text files containing shell commands are run which initialize the shell environment.

The exact scripts that run depend on how the shell was started, whether it is an interactive login shell, an interactive non-login shell, or a shell script.

Assuming the default /etc/profile, /etc/bashrc, and ~/.bash\_profile files, if you want to make a change to your user account that affects all your interactive shell prompts at startup, edit your ~/.bashrc file. For example, you could set that account's default editor to **nano** by editing the file to read:

# .bashrc

# Source global definitions

if [ -f /etc/bashrc ]; then

. /etc/bashrc

fi

# User specific environment

PATH="$HOME/.local/bin:$HOME/bin:$PATH"

export PATH

# User specific aliases and functions

**export EDITOR=nano**

Note

The best way to adjust settings that affect all user accounts is by adding a file with a name ending in .sh containing the changes to the /etc/profile.d directory. To do this, you need to be logged in as the root user.

Unsetting and Unexporting Variables

To unset and unexport a variable entirely, use the **unset** command:

**[user@host ~]$ echo $file1**

/tmp/tmp.z9pXW0HqcC

**[user@host ~]$ unset file1**

**[user@host ~]$ echo $file1**

**[user@host ~]$**

To unexport a variable without unsetting it, use the **export -n** command:

**[user@host ~]$ export -n PS1**

References

**bash**(1), **env**(1), and **builtins**(1) man pages

Guided Exercise: Changing the Shell Environment

In this exercise, you will use shell variables and variable expansion to run commands and set an environment variable to adjust the default editor for new shells.

**Outcomes:**

You should be able to:

* Edit user profile.
* Create a shell variable.
* Create an environment variable.

Log in to workstation as student using student as the password.

On workstation, run the **lab edit-shell start** command. This script verifies that the target server is running.

**[student@workstation ~]$ lab edit-shell start**

1. Change the student user's PS1 shell variable to [\u@\h \t \w]$ (remember to put the value of PS1 in quotes and put in a trailing space after the dollar sign). This will add the time to the prompt.
   1. On workstation, use the **ssh** command to log in to servera.

**[student@workstation ~]$ ssh student@servera**

*...output omitted...*

[student@servera ~]$

* 1. Use Vim to edit the ~/.bashrc configuration file.

**[student@servera ~]$ vim ~/.bashrc**

* 1. Add the PS1 shell variable and its value to the ~/.bashrc file. Remember to include a trailing space at the end of the value that you set and put the entire value in quotes, including the trailing space.

*...output omitted...*

# User specific environment and startup programs

PATH="$HOME/.local/bin:$HOME/bin:$PATH"

**PS1='[\u@\h \t \w]$ '**

export PATH

* 1. Exit from servera and log in again using the **ssh** command to update the command prompt.

**[student@servera ~]$ exit**

logout

Connection to servera closed.

**[student@workstation ~]$ ssh student@servera**

*...output omitted...*

[student@servera 14:45:05 ~]$

1. Assign a value to a local shell variable. Variable names can contain uppercase or lowercase letters, digits, and the underscore character. Retrieve the variable value.
   1. Create a new variable called file with a value of tmp.zdkei083. The tmp.zdkei083 file exists in the student home directory.

**[student@servera 14:47:05 ~]$ file=tmp.zdkei083**

* 1. Retrieve the value of the file variable.

**[student@servera 14:48:35 ~]$ echo $file**

tmp.zdkei083

* 1. Use the variable name file and the **ls -l** command to list the tmp.zdkei083 file. Use the **rm** command and the file variable name to delete the tmp.zdkei083 file. Confirm it has been deleted.

**[student@servera 14:59:07 ~]$ ls -l $file**

-rw-rw-r--. 1 student student 0 Jan 23 14:59 tmp.zdkei083

**[student@servera 14:59:10 ~]$ rm $file**

**[student@servera 14:59:15 ~]$ ls -l $file**

ls: cannot access 'tmp.zdkei083': No such file or directory

1. Assign a value to the editor variable. Use one command to make the variable an environment variable.

**[student@servera 14:46:40 ~]$ export EDITOR=vim**

**[student@servera 14:46:55 ~]$ echo $EDITOR**

vim

Exit from servera.

**[student@servera 14:47:11 ~]$ exit**

logout

Connection to servera closed.

[student@workstation ~]$

**Finish**

On workstation, run the **lab edit-shell finish** script to complete this exercise.

**[student@workstation ~]$ lab edit-shell finish**

This concludes the guided exercise.

Summary

In this chapter, you learned:

* Running programs, or processes, have three standard communication channels, standard input, standard output, and standard error.
* You can use I/O redirection to read standard input from a file or write the output or errors from a process to a file.
* Pipelines can be used to connect standard output from one process to standard input of another process, and can be used to format output or build complex commands.
* You should know how to use at least one command-line text editor, and Vim is generally installed.
* Shell variables can help you run commands and are unique to a particular shell session.
* Environment variables can help you configure the behavior of the shell or the processes it starts.