Introduction

**[A Brief Overview](https://launchschool.com/books/command_line/read/introduction" \l "briefoverview)**

We communicate with modern computers by clicking, tapping, sliding, hovering, typing, shaking, and speaking. Through simple gestures, we are able to command the computer. Many computer operating systems such as Windows and OS X provide a graphical user interface that enables us to open and view files, play games, turn on music, etc., all by simple movements and clicks of the mouse. Most operating systems that run on servers, desktops, and laptops provide another way of commanding your computer that many people have never even heard of. It is aptly called the command line interface (or CLI), the command line, or the command prompt.

Like a graphical user interface, the command line is one way that the operating system represents the computer's files, directories, and programs (which are also files) to the user. In fact, the command line is a text-based interface through which one can navigate, create, execute, and act on a computer's files and directories with precision.

What We'll Cover in this Book

Since the Linux operating system is the most commonly used operating system for servers, we will cover its command line interface in this book. Other operating systems such as Mac OS X and Unix are very similar, so you'll be able to use everything you learn in this book on those operating systems. Other operating systems, such as Windows, have command prompts, but they differ in some significant ways that make it difficult to cover their usage in one book.

How to Use the Command Line

Using the command line is as simple as opening up Terminal (available on Mac and Linux) or iTerm (available on Mac only) on your computer, typing a command and pressing enter. Most commands will return some kind of output. Go ahead and open up Terminal and try typing the following command:

$ echo "Hello World"

You should see something like this:

$ echo "Hello World"

Hello World

$

There's nothing spectacular there, but the above exercise illustrates how you can command the computer to do something and then see its result.

**[What's in it for me?](https://launchschool.com/books/command_line/read/introduction" \l "whyshouldilearncli)**

While it may seem difficult at first, being able to use the command line will empower you as a computer user. Once you've learned the basics of using the CLI, you'll be able to simplify and speed up many tasks that were previously tedious. You can be very precise about how you want your computer to perform certain tasks, or about which pieces of information it should display. By reading this book and doing its exercises, you'll gain the following knowledge and skills:

* Understand commands that tutorials tell you to use
  + As you learn programming, you will inevitably have to use the command line to install software, compile or run code, and perform other types of system administration.
  + Learning the basics of the command line will help you understand why you are running the commands, and give you a general idea of how they work.
* Power over the computer
  + Being able to communicate with your computer or server via command line gives you immense and precise power over your computer.
  + With the command line, you can easily monitor your computer and its resources.
  + You can take advantage of a lot of powerful tools in their raw, powerful form.
  + You can learn to diagnose issues with your own software and others' software.

Preparations

**[Who should read this book](https://launchschool.com/books/command_line/read/preparations" \l "whoshouldreadthisbook)**

This book is intended to help you on your path to becoming a skilled programmer. It covers the basics of command line usage in a Unix-like environment, which includes Linux and Mac OS X. It *does not* include Windows PowerShell or Windows Command Prompt usage. It is assumed that you know the basics of using a computer and have run a command on the command line before. Knowledge of programming is not required.

**[Getting the Most Out of this Book](https://launchschool.com/books/command_line/read/preparations" \l "gettingthemostoutofthebook)**

There are a few things you must do to successfully learn the lessons in this book. First, try things out and make what you've learned your own. Don't want to mess up your own computer or server? No problem: you can easily set up a separate server or virtual machine to mess around with. See the next section for some suggestions. Second, do the exercises. They are there to give you a real-world understanding of the command line and will help you remember what you've learned. Third, explain what you've learned to a friend or coworker, or to a [rubber duck](http://en.wikipedia.org/wiki/Rubber_duck_debugging) if that's more comfortable.

**[Server Setup](https://launchschool.com/books/command_line/read/preparations" \l "serversetup)**

If you don't want to mess up your computer's console while going through the examples in this book, you can set up a server using one of many online cloud services or set up your own virtual machine. The following are some of the available options:

1. Sign up for a service that provides a full, in-browser development environment, such as [Cloud 9](https://c9.io/). Once you've set up an account, you can ignore everything except the terminal or console. We recommend this option if you're new to the command line, as it takes the least amount of effort to set up.
2. Use [Rackspace](https://cart.rackspace.com/en-us/cloud) or [Digital Ocean](https://cloud.digitalocean.com/registrations/new) to set up a managed server that provides console access. For this option, you'll need a little bit of experience with servers.
3. Set up a virtual machine on your computer. You can use [Virtualbox](https://www.virtualbox.org/), [VMWare](http://www.vmware.com/), [Vagrant](https://www.vagrantup.com/), [Docker](https://www.docker.com/), or a combination of these. Setting up a virtual machine is out of the scope of this book, but a quick Google search should be enough to get you going. You should have a lot of hands-on experience with configuring software and in-depth knowledge of computers before taking this option.

The Command Line Interface

**[What is an interface?](https://launchschool.com/books/command_line/read/command_line_interface" \l "whatisaninterface)**

According to the online Merriam Webster dictionary, an interface is "a system that is used for operating a computer: a system that controls the way information is shown to a computer user and the way the user is able to work with the computer."[1](http://www.merriam-webster.com/dictionary/interface) As this definition explains, an interface has two parts: 1) a display of information about what the computer is doing and 2) a method for telling the computer what to do. The command line interface is a very basic interface that uses mostly text for both the display and input components of the interface. In a nutshell, the command line allows for text-based communication with a computer. Let's look at the two parts of the command line interface: the display and the input.

CLI Display

The command line can be the default interface for a computer, but most personal computers use a program (like Terminal) within the desktop graphical interface to provide the command line interface. Let's dissect the CLI's language for a moment. To follow along, log in to your virtual machine or server.

Below is an example of the prompt on an Ubuntu Linux server:

ubuntu@chopin:~$

The above prompt follows this format:

[user]@[hostname]:[current\_directory]$

The [user] portion ("ubuntu") represents the current user that is logged in to the command line interface. The [hostname] portion represents the computer's name. Following those two pieces is :[current\_directory]. The colon is just for separation, and the [current\_directory] displays the path of the directory that you are in. If you've just logged in, it's probably just a tilde (~), which represents the home directory. The last piece is $ (note the trailing space, which refers to the extra space following $). This whole piece of text is called the prompt, or PS1. It can be modified to fit your needs, but usually displays some very basic information that shows you the context of what you're doing. Whenever you log in to a server, or whenever you open up Terminal on your own computer, you'll be presented with a prompt like this one. For the rest of this book, we'll represent the prompt as a simple $.

At the end of the prompt, you should see a cursor which, in the world of the command line, is just a box that blinks on for a second and off for a second. The cursor shows where you are able to input additional text. The thing that allows you to input text is called STDIN (standard input).

The last piece of the display portion of the CLI is the output of your commands. As you can see in the image below, when you type a command, it may have textual output:

$ ls /

bin home lib64 opt sbin usr

boot initrd.img lost+found proc srv var

dev initrd.img.old media root sys vmlinuz

etc lib mnt run tmp vmlinuz.old

When the output has printed to the screen and the command or program exits, the prompt is displayed again below the output. To review, the components of the CLI display are the **prompt**, the **cursor**, the **input** (text you have typed in), and the **output** of your commands and programs.

CLI Input

The second component of this textual interface is the input. By typing text into the command line interface, you are creating input for the CLI to interpret and act on. Using commands such as echo or ls, you can command the computer to do your bidding. Let's try a command. Try typing echo "Hello World" into the console:

$ echo "Hello World"

The echo command is very simple, but it can also be very useful. All it does is send text to the CLI's output. Another useful command is the pwd command. If you want to know where you are in your computer's file system, you can type pwd and press enter to execute it. You should get something like the following as output from that command:

$ pwd

/home/ubuntu

The commands above are simple, and don't actually make any changes to your command line environment, the computer, or its files -- they only display some output. But other commands such as rm, cp, and source can directly affect the files on your computer or can change your command line environment.

The Anatomy of a Command

Commands come in all shapes and sizes, but they have many features in common. When you type a command into a terminal, it will always take the following format:

[command] [arguments...]

A command can be the path to a file (e.g. /path/to/file), or it can be a command that your terminal is already aware of (e.g. echo). The second portion of the command usually consists of what are called "arguments". Arguments are strings that are passed to the program that you are executing. Stated more simply, these are pieces of information that you are providing to your command.

To better understand how this works, we can compare the command line and its commands to a well-trained army. In this army, each soldier has a very specific duty, and is trained to do it with perfection. Some duties, like cleaning the dishes, don't take any extra information for them to be accomplished successfully. Other duties require, or at least benefit from, extra information. If you are the general and you tell a soldier to go on patrol, he may be able to go out and patrol somewhere, but it may or may not be where you need him most. He'll probably just go to his usual patrolling area and start there.

Command line commands are basically the same. If you type a command by itself, it will sometimes execute just fine, using default information to do its job. Other times, however, you really need to give the command more information so that it can know exactly how to perform its duty. Take, for example, the tar command. If you open the command line prompt, type tar, and press enter, it won't do much. In fact, all it does is ask you for more information: "You want me to patrol? Where should I patrol?"

$ tar

tar: You must specify one of the '-Acdtrux', '--delete' or '--test-label' options

Try 'tar --help' or 'tar --usage' for more information.

The tar command is an archival command that can archive, compress, and extract files, but to make it work, you need to tell it which files to take action on, and what exactly to do with those files. You should also tell the command where to put the extracted or compressed files once it's done. Let's look at the command in action (the directory is fictitious, so you won't be able to execute this command as-is; also, the lines that start with # are comments):

# Command: compress the files-to-archive directory to archive.tgz in this

# directory.

$ tar -c -z -f ./archive.tgz ./files-to-archive/

In the above example, tar is the command, and -c, -z, -f, ./archive.tgz, ./files-to-archive/ are the arguments. Each of the first three arguments is a flag, which is a special type of argument. The c stands for "create", the z stands for "zip", and the f stands for "file". Flags can take different forms, but they usually start with a dash (-) for abbreviated flags and two dashes (--) for full word flags (e.g. --format). Abbreviated flags can usually be combined, so the above could also be written -czf. To find out which flags a command uses and what they mean, you can type man ("man" is short for "manual") followed by the name of the command:

$ man tar

TAR(1) BSD General Commands Manual TAR(1)

NAME

tar — The GNU version of the tar archiving utility

SYNOPSIS

tar [-] A --catenate --concatenate | c --create | d

--diff --compare | --delete | r --append | t

--list | --test-label | u --update | x --extract

--get [options] [pathname ...]

DESCRIPTION

Tar stores and extracts files from a tape or disk ar‐

chive.

...

Type q to exit man.

**[What is the Command Line used for?](https://launchschool.com/books/command_line/read/command_line_interface" \l "whatiscliusedfor)**

Now that we've discussed what an interface is, and how the command line is an interface all on its own, let's look at what the command line is typically used for.

The most common use of the command line is what's called "system administration" or, basically, managing computers and servers. This includes installing and configuring software, monitoring computer resources, setting up web servers, and automating processes. The following is a list of common tasks for programmers:

* Restart servers
* Rename hundreds or thousands of files according to a prescribed pattern
* Manage system logs
* Set up scheduled tasks (cron jobs)
* Debug server issues
* Monkey patch code on a server
* Query data
* Set up databases and servers

... and many more.

**[What are some common commands?](https://launchschool.com/books/command_line/read/command_line_interface" \l "commoncommands)**

The following is a list of some of the most common commands used on the command line. The exercises will walk you through how to use each of them.

| **Command** | **Description** |
| --- | --- |
| cd | Change directory. |
| ls | List files and directories in current directory. |
| pwd | Display the path of the current directory. |
| touch | Create a file. |
| mkdir | Create a directory. |
| rm | Remove a file or directory. Warning: deleting a file or directory with this command is permanent! |
| cp | Copy a file or directory. |
| mv | Move or rename a file or directory. |
| echo | Print text to STDOUT. |
| cat | Display contents of a file. |
| more | Display contents of a file, starting at the top and letting the user scroll down. |
| less | Display contents of a file in an even more interactive way. |
| head | Display the first part of a file. |
| tail | Display the last part of a file. |
| man | Display documentation about a command. |

**Exercises**

1. The echo command has only one optional flag: -n, which means "Do not print the trailing newline character." Experiment with echo. Run the following commands, guessing before running each what the output will be:
2. $ echo "hello world"
3. $ echo hello world
4. $ echo -n hello world
5. $ echo hello world -n
6. $ echo "hello world" -n
7. $ echo "-n hello world"

**Solution**

1. We'll talk more about navigating the command line in the next chapter, but as a bit of a preview, let's go over the following commands: cd, ls, and pwd.

The cd command helps you change your current directory. The ls command gives you information about what files and directories are in a certain directory. Finally, the pwd command, as mentioned above, tells you what your current directory is.

First, let's see what your current directory is:

$ pwd

/home/ubuntu

You should get something like /home/ubuntu as output. So, all the pwd command does is output a string of characters that represent the directory that you are currently in.

Next, you can try changing your directory with the cd command:

$ cd /

$ pwd

/

The cd command, as can be seen above, doesn't have any output. Now when you run pwd, it shows that you are in the /, or root directory. If you try to change your current directory to a location that doesn't exist, cd will output an error:

cd /asdf

-bash: cd: /asdf: No such file or directory

Now let's see what files and directories exist in the root directory:

$ ls

bin home lib64 opt sbin usr

boot initrd.img lost+found proc srv var

dev initrd.img.old media root sys vmlinuz

etc lib mnt run tmp vmlinuz.old

Without any arguments, ls just prints out a list of the files and directories in the current directory. It doesn't tell you much about those files and directories, though. If you add a couple of arguments, you can get the ls command to give up more information about each item in the directory:

$ ls -lah

total 84K

drwxr-xr-x 22 root root 4.0K May 14 17:31 .

drwxr-xr-x 22 root root 4.0K May 14 17:31 ..

drwxr-xr-x 2 root root 4.0K May 12 23:20 bin

drwxr-xr-x 3 root root 4.0K May 12 23:20 boot

drwxr-xr-x 13 root root 3.9K May 14 17:31 dev

drwxr-xr-x 96 root root 4.0K Jun 22 17:15 etc

drwxr-xr-x 3 root root 4.0K May 12 23:19 home

lrwxrwxrwx 1 root root 33 May 12 23:20 initrd.img -> boot/initrd.img-3.13.0-52-generic

lrwxrwxrwx 1 root root 33 Mar 25 11:51 initrd.img.old -> boot/initrd.img-3.13.0-48-generic

drwxr-xr-x 21 root root 4.0K May 13 19:03 lib

drwxr-xr-x 2 root root 4.0K Mar 25 11:50 lib64

drwx------ 2 root root 16K Mar 25 11:53 lost+found

drwxr-xr-x 2 root root 4.0K Mar 25 11:50 media

drwxr-xr-x 2 root root 4.0K Apr 10 2014 mnt

drwxr-xr-x 2 root root 4.0K Mar 25 11:50 opt

dr-xr-xr-x 127 root root 0 May 14 17:30 proc

drwx------ 3 root root 4.0K May 13 19:00 root

drwxr-xr-x 18 root root 700 Jul 1 14:12 run

drwxr-xr-x 2 root root 4.0K May 12 23:20 sbin

drwxr-xr-x 4 ubuntu ubuntu 4.0K Jun 22 17:06 srv

dr-xr-xr-x 13 root root 0 May 14 17:30 sys

drwxrwxrwt 4 root root 4.0K Jul 1 14:40 tmp

drwxr-xr-x 10 root root 4.0K Mar 25 11:50 usr

drwxr-xr-x 12 root root 4.0K Mar 25 11:52 var

lrwxrwxrwx 1 root root 30 May 12 23:20 vmlinuz -> boot/vmlinuz-3.13.0-52-generic

lrwxrwxrwx 1 root root 30 Mar 25 11:51 vmlinuz.old -> boot/vmlinuz-3.13.0-48-generic

We'll go into detail of what everything means in later chapters, but for now, you can see that adding the arguments -l, -a, and -h (grouped into one argument above), adds a lot more information and formats the items into a more easy-to-read list.

1. The command line offers a bunch of commands to work with files and directories. This exercise will show the basic usage of the following: touch, mkdir, mv, cp, and rm.

First, let's make sure we're in your home directory (recall that ~ stands for your home directory):

$ cd ~

$ pwd

/home/ubuntu

Now, let's create a practice directory to mess around with:

$ # Make a directory called "practice"

$ mkdir practice

$ ls

practice

You can see that now there's a directory named "practice" in your home directory. Let's change our current directory to the "practice" directory that we just created:

$ cd practice

Now, in this directory we can create new files, move or rename them, copy them, and remove them. After that, we'll remove the whole "practice" directory to clean up.

$ # Create an empty file and verify that it got created

$ touch example.txt

$ ls

example.txt

The touch command created the empty file "example.txt" in the current directory. You can move or rename a file with the mv command:

$ # Rename example.txt to example1.txt

$ mv example.txt example1.txt

$ ls

example1.txt

$ # Make another directory

$ mkdir tmp

$ # Move example1.txt to the new "tmp" directory

$ mv example1.txt tmp/

$ ls tmp/

example1.txt

$ ls

tmp

$ # Move it back and rename it

$ mv tmp/example1.txt example2.txt

$ ls

example2.txt tmp

You can see from the above examples that moving or renaming a file is basically the same thing on the command line, and follows this pattern:

$ mv [source] [destination]

Now, let's remove the example file and then the whole practice folder:

$ rm example2.txt

$ ls

tmp

To remove a folder and all its contents, you need to specify the -r (recursive) option.

$ cd ..

$ ls

practice

$ rm -r practice

$ ls

The practice folder (and all its contents) are now gone.

Warning: using the rm command is dangerous and permanent. Do not issue this command until you know for certain you are deleting the right file. Using the rm -r command is even more dangerous, as it will delete recursively.

1. One of the most common tasks when using the command line interface is reading the contents of a file. This exercise will go over some common commands for doing so: cat, more, less, head, and tail.

To print out all the contents of a file, use cat:

$ cat file.txt

All the content of file.txt printed out here.

To print out the first few lines of a file, use head:

$ head /etc/mime.types

#

# MIME media types and the extensions that represent them.

#

# The format of this file is a media type on the left and zero or more

# filename extensions on the right. Programs using this file will map

# files ending with those extensions to the associated type.

#

# This file is part of the "mime-support" package. Please report a bug using

# the "reportbug" command of the "reportbug" package if you would like new

To print out the last few lines of a file, use tail:

$ tail /etc/mime.types

video/x-ms-wmx wmx

video/x-ms-wvx wvx

video/x-msvideo avi

video/x-sgi-movie movie

video/x-matroska mpv mkv

x-conference/x-cooltalk ice

x-epoc/x-sisx-app sisx

x-world/x-vrml

To print out the contents of a file, but only fill one screen's worth at a time, use more:

$ more /etc/mime.types

# The first page of /etc/mime.types will show up here, and you can use the

# down arrow to go to the next line, or the space bar to go to the next

# page.

Use less when you need to navigate backward and forward in a file:

$ less /etc/mime.types

The less command allows you to go forward one line with the down arrow, backward one line with the up arrow, and backward and forward a page with the page up and page down keys. You can also use the space bar in the same way you can with the more command.

Note: to exit more or less, type the q key.

1. You can get more information about what a command does, how it works, and which flags you can use by referencing the manual for that command. To read the manual pages (typically called manpages) for a command, use the man command:
2. $ man touch
3. TOUCH(1) User Commands TOUCH(1)
4. NAME
5. touch - change file timestamps
6. SYNOPSIS
7. touch [OPTION]... FILE...
8. DESCRIPTION
9. Update the access and modification times of each FILE to the
10. current time.
11. ...

The manual pages will sometimes have the available flags and the possible orders they can be used in the "SYNOPSIS" section. In the case of the touch command, the flags can be used in any order just after the touch command itself, and the file is the last argument. The possible options are described in the "DESCRIPTION" section.

You may have noticed that the man command uses less to display the content of the manpages, so you can use the up- and down-arrow keys to navigate the contents of the manual.

To exit man, you can type the q key.

Files, Directories, and Executables

**[The Core of the Command Line](https://launchschool.com/books/command_line/read/files_directories_executables" \l "clicore)**

In a desktop environment, like that of OS X, you have windows, menu bars, and the desktop to give context to what you are doing. In the command line, however, the context is solely the file system. In fact, files and directories are what make up the command line. Almost everything you do at the prompt will deal with files. Every time you type a command, you are telling the computer to execute a file.

Not only does the file system provide context for you when you're trying to work on or get information about files and directories, it also provides context for the commands you run. For example, you can use the ls command to list the files in a directory. When you run the command by itself, it uses your current directory as context, and lists the files that are in the directory you are in.

**Linux/Unix File System Legend**

Let's look for a moment at some of the symbols that will help us navigate the command line:

* / - The root directory or a separator when listing directories
* . - The current directory (also ./) or the same level
* .. - The directory one level up (also ../)
* ../.. - Two levels up
* ~ - Your "home" directory, or the directory you are placed in when you log in.
* \* - The "splat" or "glob" operator. This is the wildcard of the command line and represents "any characters."

The above symbols can be combined with directory and file names to represent their locations. The path /home/ubuntu can be dissected as follows:

root directory + "home" directory + directory separator + "ubuntu" (user)

directory

While it may not make sense to do so, somewhat "nonsensical" combinations can represent valid paths. The following path is the same as /home/ubuntu:

/home/ubuntu/../.././home/ubuntu/

Let's break the above path into its parts:

* /home/ubuntu/ - The ubuntu directory within the home directory within the root (/) directory.
* ../../ - Up two directories (which takes us back to the root directory).
* ./ - The same directory (which is still /).
* home/ubuntu/ - Back down into the home directory, then the ubuntu directory.

Note that there's a dramatic difference between a path that starts with a leading slash vs one that doesn't. For example:

* /home/ubuntu This path specifies a folder called "ubuntu" that lives under your root directory, then home directory.
* home/ubuntu This path specifies a completely different folder. This path means there's a folder called "ubuntu" that lives in a directory called "home", which is itself in the *current directory*.

The leading slash makes all the difference. Make sure to pay careful attention to that whenever you're looking at paths.

**[Navigating](https://launchschool.com/books/command_line/read/files_directories_executables" \l "navigating)**

Navigating your computer's file system is pretty easy with the help of the File System Legend above, and a few simple commands:

* cd - change directory
* ls - list files
* pwd - display the current working directory

Let's get a feel for navigating the command line interface by opening up Terminal or logging in to your managed server or virtual machine and treading water for a bit. After logging in, type the following commands and note what happens after each step:

# Change directory (with no arguments)

$ cd

$ pwd

/home/ubuntu

# Change to root directory

$ cd /

$ pwd

/

# Change directory (with no arguments again)

$ cd

$ pwd

/home/ubuntu

# Go up one directory

$ cd ../

$ pwd

/home

# Show files and directories (with no arguments, it uses current directory)

$ ls

ubuntu

# Navigate into "ubuntu" directory

$ cd ubuntu

$ pwd

/home/ubuntu

# Change to ~ (home) directory

$ cd ~

$ pwd

/home/ubuntu

# Show files and directories in root directory

$ ls /

bin boot dev etc home lib lib64 media mnt opt proc root rootfs run sbin selinux srv sys tmp usr var

# Show files that match a certain pattern

# using the splat operator (asterisk)

# The following example may not produce the same

# results on your system. On some systems, such as

# Ubuntu, you may need to use sudo ls /\*ot instead

# of ls /\*ot, but on others, such as a Mac, you

# will only get error messages. What's important

# here is to learn how to use the commands, not

# replicate the results.

$ ls /\*ot

boot root

Did you notice that typing cd by itself takes you to your "home" directory? Which command(s) made you "descend" into a directory? Which command(s) made you go up a directory? What was the difference in the output of ls vs. ls /? What do you think the output of ls ./ would be if you first ran cd /?

The last example shows that the splat (\*) allows you to list files that match a pattern. The asterisk is the wildcard and represents any number of any characters, so \*ot represents any file or directory that ends in ot.

**[Managing Files and Folders](https://launchschool.com/books/command_line/read/files_directories_executables" \l "managingfiles)**

With an understanding that "where you are" in the command line provides context for your commands, and a working knowledge of navigating the command line interface, we can move on to learning to manage files and directories from the command line.

When you open up Finder in Mac or Windows Explorer in Windows, you can see manila-folder-style directories along with icons that represent documents, images, and programs. Think for a moment about some of the things that you do with the files from time to time. Do you ever move files around? Maybe sometimes you copy a file so that you can make a new version. Have you made new directories before? It's probably not too surprising by now that you can do the same things from the command line. Some of the most frequently used commands for managing files and directories on the command line are cp, mv, mkdir, touch, and rm. Their meanings are listed here:

| **Command** | **What it does** |
| --- | --- |
| cp | Copy one or more files to a new location |
| mv | Move or rename a file or directory |
| mkdir | Make a new directory |
| touch | Create a new file or update modification time if a file with that name exists |
| rm | Remove one or more files or directories |

To get acquainted with these commands, go ahead and try them out. Try the following exercises to solidify your understanding of the above commands:

1. Create a directory in your home directory called "cli-tmp".

**Solution**

1. From the home directory, create a file in the cli-tmp directory called "from-home.txt".

**Solution**

1. Navigate to the cli-tmp directory, then create a file called "in-cli-tmp".

**Solution**

1. Try to make a directory called "in-cli-tmp" within the cli-tmp directory. What happens?

**Solution**

1. Remove the from-home.txt file.

**Solution**

1. Remove the cli-tmp directory (hint use man to see how to remove a directory recursively).

**Solution**

1. Create a nested set of directories in your home directory: cli-tmp > parents > children > grandchildren (hint: use the -p flag to do it all at once).

**Solution**

1. Navigate to the children directory.

**Solution**

1. Create a file named "bob".

**Solution**

1. Move the file named "bob" to the grandchildren directory.

**Solution**

1. Copy the grandchildren directory to the parents directory, and rename it "nephews".

**Solution**

1. Copy the contents of the "nephews" directory to the "children" directory.

**Solution**

1. View what you've done with the tree command: $ tree ~/cli-tmp.

**Solution**

1. Remove the "bob" file from the grandchildren directory.

**Solution**

1. Remove the cli-tmp directory.

**Solution**

**[Executables](https://launchschool.com/books/command_line/read/files_directories_executables" \l "executables)**

In the last chapter, we talked about the anatomy of commands, which was basically "command plus arguments." But what is a command really? As mentioned at the beginning of this chapter, a command is just a file. That's right: ls, mkdir, and cd are all files. Files that can be used as commands are called executables. Not all files are executables.

What makes an executable different from other files?

1. They have special characters at the beginning to tell the computer how to execute them.
2. They have scripts or machine language as their content.
3. They have the executable permission (we'll talk more about this later).

To run an executable, you just type its path as the first part of your input, then type in your arguments, and hit enter, like this:

$ /bin/echo "Hello World"

Hello World

In the above example, /bin/echo is the path to the executable, and "Hello World" is the first (and only) argument.

If you want to execute a file in your current directory, you have to type ./ or the path to your current directory before the filename. For common or installed commands, you won't usually have to specify the path of the executable because the command line interface already knows where to find it. That's why you can type echo instead of /bin/echo.

When you use the echo executable, it will probably take no more than a few milliseconds to run, and the chances that it will freeze or get stuck are almost zero. Some executables, however, cause long-running processes, and it is useful at times to be able to exit out of them before they exit on their own. Some commands and executables run in a loop, and won't exit until you tell them to. Others freeze or take a lot longer than you expected, and need to be stopped short. On Unix-like systems, there is an almost ubiquitous way of exiting out of any executable's process: Ctrl + c. This key combination sends a signal to the process to terminate SIGINT.

If you are using Bash commands and accidentally create an [infinite loop](https://en.m.wikipedia.org/wiki/Infinite_loop), you can exit out of it with Ctrl + c:

$ while true; do echo 'Hit CTRL+C'; sleep 1; done

Hit CTRL+C

Hit CTRL+C

Hit CTRL+C

The above will echo "Hit CTRL+C", then sleep for a second, until you hit Ctrl + c on your keyboard.

Note the format "Ctrl + c" means "hold down the control key, then press c, and release." You'll see other combos like this as you learn more about programming and about using Bash.

In the next chapter, we'll discuss how the command line knows where installed commands are located.

**Exercises**

1. List the hidden files in a directory.

**Solution**

1. List the files in the parent directory. Then list the files in the parent directory's parent directory.

**Solution**

1. What's the difference between abc/ and /abc?

**Solution**

1. What does ../abc mean?

**Solution**

1. Suppose you are in a directory with 7 files. You need to move 6 of them into a "temp" directory, that you have yet to create. How do you do that?

**Solution**

1. Suppose you have two directories called xyz/ and abc/. How do you move all the files in abc/ that end with ".txt" into xyz/?

**Solution**

1. Let's review some common navigation tasks:

To get your current location, use pwd:

$ pwd

/home/ubuntu

To change your current directory, use cd. Using it without any arguments will take you to your home directory:

$ cd

$ pwd

/home/ubuntu

You can also pass the $HOME variable as an argument, or use the ~ (tilde) as the first argument to achieve the same results:

$ cd $HOME

$ pwd

/home/ubuntu

$ cd ~

$ pwd

/home/ubuntu

Go to the root directory (/):

$ cd /

$ pwd

/

Use the ls command to determine what files and directories are located in the current directory:

$ cd /

$ pwd

/

$ ls

bin dev home lib64 lost+found mnt proc run selinux sys usr

boot etc lib local media opt root sbin srv tmp var

Pass a directory or a path as an argument to the cd command to go directly to that location:

$ cd usr

$ pwd

/usr

You can also get a sneak peek into what directories are located in a specific path:

$ cd /

$ pwd

/

$ ls usr

bin etc games include lib lib64 libexec local sbin share src tmp

Use the -lah set of flags as arguments to the ls command to get more detailed information about files and directories:

$ cd /

$ ls -lah

total 112K

dr-xr-xr-x 24 root root 4.0K Mar 16 20:56 .

dr-xr-xr-x 24 root root 4.0K Mar 16 20:56 ..

-rw-r--r-- 1 root root 0 Mar 16 20:56 .autofsck

dr-xr-xr-x 2 root root 4.0K Oct 1 2014 bin

dr-xr-xr-x 3 root root 4.0K Oct 1 2014 boot

drwxr-xr-x 16 root root 2.8K Jun 3 21:48 dev

drwxr-xr-x 76 root root 4.0K Jun 3 21:48 etc

drwxr-xr-x 3 root root 4.0K Sep 29 2014 home

dr-xr-xr-x 7 root root 4.0K Mar 25 2014 lib

dr-xr-xr-x 10 root root 12K Oct 1 2014 lib64

drwxr-xr-x 2 root root 4.0K Mar 25 2014 local

drwx------ 2 root root 16K Mar 25 2014 lost+found

drwxr-xr-x 2 root root 4.0K Jan 6 2012 media

drwxr-xr-x 2 root root 4.0K Jan 6 2012 mnt

drwxr-xr-x 3 root root 4.0K Mar 25 2014 opt

dr-xr-xr-x 75 root root 0 Mar 16 20:56 proc

dr-xr-x--- 3 root root 4.0K Jun 3 21:48 root

drwxr-xr-x 4 root root 4.0K Sep 29 2014 run

dr-xr-xr-x 2 root root 12K Oct 1 2014 sbin

drwxr-xr-x 2 root root 4.0K Jan 6 2012 selinux

drwxr-xr-x 2 root root 4.0K Jan 6 2012 srv

dr-xr-xr-x 13 root root 0 Mar 16 20:56 sys

drwxrwxrwt 3 root root 4.0K Jun 10 03:34 tmp

drwxr-xr-x 13 root root 4.0K Mar 25 2014 usr

drwxr-xr-x 19 root root 4.0K Mar 25 2014 var

1. In this exercise we're going to play with executables. We'll find the location of so-called "default executables" by using the which command, then execute a file using its absolute path.

To figure out a default executable's location, you can use the which command:

$ which touch

/bin/touch

Now we can execute the touch command using it's full path or its name:

$ cd

$ touch test

$ ls

test

$ /bin/touch test2

$ ls

test test2

We can do the same thing now with the rm command:

**Warning!** The "rm" command is extremely dangerous. There is no easy, or even moderately difficult, way to restore what you delete with the "rm" command!

$ cd

$ rm test

$ ls

test2

$ which rm

/bin/rm

$ /bin/rm test2

$ ls

$

The Environment

**[Environment Variables](https://launchschool.com/books/command_line/read/environment" \l "environmentvariables)**

In the last chapter, we talked about how your "current directory" provides context for commands you run. Another way of providing context is through something called **environment variables**. In programming, variables are used to store data and to be able to reference and retrieve that data at a later point using a name. In the command cd $HOME, the $HOME part is a reference to the HOME variable, and is replaced by the path to your home directory when the command is run. In other words, running cd $HOME is the same as running cd /home/ubuntu, assuming your home directory is /home/ubuntu.

When you log in to the command line, a variety of environment variables are automatically set. You can see exactly what variables have been set, along with their values, by running env at the command line. Type env, hit enter, and find the value for HOME. It should say something like /home/ubuntu, where ubuntu will be replaced by your username. If you're doing this on a Mac, the value will probably be something like /Users/bob. This is the path to your home directory.

**[How to Change your Command Line Environment](https://launchschool.com/books/command_line/read/environment" \l "changeenvironment)**

While there are several environment variables that are set for you automatically, you can also set your own or modify existing variables. You can do this on the fly, so that your changes only affect the current command or the current session, or you can make the changes more permanent so that they stick between sessions.

**Note:** The term "session" refers to the state of being logged in to a computer's command line interface. When you log in, you start a new session, in which your commands will be recorded and other contextual information will be maintained. When you close Terminal or type "exit", your session is closed and that context and data is lost.

Setting Environment Variables on the Fly

There are two ways to set an environment variable on the fly:

1. Set the variable on its own line, then use it anywhere:

$ SOMETHING='some value'

$ echo $SOMETHING

some value

2. Set the variable before a command, on the same line:

$ SOMETHING='a value' env

...

SOMETHING=a value

...

Note: You cannot (very easily) use a value on the same line that you set it. That's because variables are evaluated before the setting occurs:

$ SOMETHING='something else' echo $SOMETHING

# no output

Did you notice that when you set a variable you don't prepend the dollar sign, but when you reference it, you do? Also note that there should be no spaces between the variable and the equal sign or the equal sign and the value. Lastly, it's usually best to use quotations around the value that you are assigning to the variable, but you don't have to when the value doesn't have any special characters.

Let's try changing our current session's environment. Maybe you'd like to simplify your prompt. To change your prompt, adjust the PS1 variable to whatever you'd like it to be:

$ PS1="(testprompt)> "

(testprompt)>

As you can see, the prompt is now (testprompt)>, and every time a command finishes, it will show up again. If you want a more complicated prompt, try the following:

$ PS1="\n\[\e[0;37m\][\h] \e[0;35m\]\d\e[0m\]\n\[\e[0;31m\]\u\[\e[0;34m\] in \[\e[1;33m\]\w\[\e[m\]\[\e[0;31m\]\n\[\033[35m\]$\[\033[00m\] "

[chopin] Wed Apr 08

ubuntu in ~

$

The new prompt is multi-line and has color-coding. If you want to revert to your old prompt, just close your session and start a new one. Since we made the changes to the environment variable PS1 on the fly, they won't be used in future sessions.

Making More Permanent Changes

It is possible to make more permanent changes to the command line environment. When you start a command line session by opening a new Terminal window, one or more **environment files** are executed. These files can be used to modify or create environment variables. They are usually located in your home directory and include the following files: .bashrc, .bash\_profile, .bash\_login, .login, .profile. Because they start with a ., they are considered to be "hidden" files, and using the ls command alone won't show them. Type ls -a ~ to see them listed along with other files in your home directory. Remember, the ls is the command, and the -a and ~ are arguments to the ls command. The -a flag tells the ls command to include files that start with . in its output, while the ~ is the directory that ls should inspect (recall that ~ means your "home" directory).

**Note:**

Macs don't have a default .bashrc file. If you are on Mac create one using command touch .bashrc, and then update .bash\_profile to include the following line somewhere near the top of the file:

[ -r ~/.bashrc ] && . ~/.bashrc # be careful with the spaces

If necessary, create .bash\_profile. Include those lines in .bash\_profile only if they aren't already there.

$ ls -a ~

. .cache .sudo\_as\_admin\_successful

.. .mysql\_history .vbox\_version

.bash\_history postinstall.sh .veewee\_version

.bash\_logout .profile .vim

.bashrc .ssh .viminfo

The rules behind which environment file is read for a new session are complicated and depend on how the session is created. For our purposes, using .bashrc should be sufficient. If your edits aren't working, try one of the other files.

Log in to your console and type the following command:

$ cat ~/.bashrc

The cat command reads the file and displays its contents. You should see some lines that start with export. The export command basically means "make this variable available globally," and variable assignment that starts with export in your .bashrc will be included in your environment when you log in. Note that if an environment variable has been exported once, you don't need to continually put export before it when you set its value.

If you're ready to customize your prompt a bit more permanently, open the .bashrc file in a code editor, and add your custom prompt to the bottom of your .bashrc file:

PS1="[your custom prompt goes here] "

**Note:** Editing hidden files can be a bit tricky if you've never done it before. The following steps will guide you through editing your .bashrc file.

1. Open .bashrc with Nano by entering following command:

$ nano ~/.bashrc

2. Place your cursor on the very last line of the file (make sure it's a new, blank line) and paste the following:

PS1="[your custom prompt goes here] "

3. Save .bashrc by pressing <Ctrl> + o then <Enter>. Exit Nano by pressing <Ctrl> + x.

4. Re-run .bashrc by entering the following command:

$ source ~/.bashrc

The export isn't needed because the variable is already available globally. Use the following pieces along with any custom text to make your prompt:

|  |  |
| --- | --- |
| **\h** | Hostname |
| **\u** | User name |
| **\w** | Current directory |
| **\W** | Basename of current directory |
| **\d** | Current date |
| **\n** | Newline |

Did you notice that just editing and saving your .bashrc file didn't do anything? The file is only evaluated, or run, when you first log in. If you want to re-run a particular environment file like .bashrc or .bash\_profile, use the source command:

$ source ~/.bashrc

To revert back to your old prompt, edit the same file and remove your PS1 setting. Then run source on that file:

[my custom prompt]$ source ~/.bashrc

$

Using Environment Variables

Let's look for a moment at the different ways we can use environment variables.

1. As Parts of Commands

First, variables can be used as arguments to commands. Take a look at the following example:

$ MESSAGE='Hello, world!'

$ echo $MESSAGE

Hello, world!

This is a very simple example, but you can see that the $MESSAGE variable is used as the first (and only) argument to the echo command. You can actually use variables as commands as well:

$ MESSAGE='Hello, world!'

$ COMMAND="echo"

$ $COMMAND $MESSAGE

Hello, world!

2. Interpolated in Strings

Variables can also be interpolated, or included, in other strings. Take the following example:

$ MESSAGE1="This is message 1."

$ MESSAGE2="This is message 2."

$ MESSAGE="$MESSAGE1 $MESSAGE2"

$ echo $MESSAGE

This is message 1. This is message 2.

To have a variable get interpolated, you have to use the double quotation mark ("), not the single quotes ('). Try the following example in your command line.

$ MESSAGE1="This is message 1."

$ MESSAGE2="This is message 2."

$ MESSAGE='$MESSAGE1 $MESSAGE2'

$ echo $MESSAGE

$MESSAGE1 $MESSAGE2

3. Behind the Scenes

Environment variables can be used by commands (programs) behind the scenes. In other words, you can set a variable, then run a command without passing the variable as an explicit argument to that command, and the command could use that variable. The PWD variable is automatically used by any command that tries to get the user's current directory. The HOME variable is automatically used by cd when you don't pass any arguments to it. If you make up a custom variable (like PI=3.14), only programs that know about it will be able to use it without explicitly using it as an argument.

If you want to temporarily change a variable before it gets used in a command behind the scenes, you can set the variable immediately preceding the command on the same line:

# Set home to root directory and change to home.

$ HOME=/ cd

$ pwd

/

# Change to home directory.

$ cd

$ pwd

/home/ubuntu

Note how the second cd takes you to your original home directory, whereas the first cd takes you to the root directory because that's what you set HOME to.

**[$PATH and Executables](https://launchschool.com/books/command_line/read/environment" \l "path)**

One of the most important environment variables you'll work with on the command line is PATH. In the last chapter, we discussed how commands are really just files, but we didn't talk about how the command line knew which file to execute for commands like cd or echo or other built-in or installed programs. The PATH variable provides the additional context that the command line needs to figure out which particular file to execute. Let's look at a PATH variable's value:

$ echo $PATH

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games

If you examine the output of the echo $PATH command above, you'll see that it is a bunch of paths connected by colons. You may have noticed that most of the paths end in /bin. This is because bin is short for "binary", and bin is a standard directory name for executable files, or programs.

Let's look at what resides in one of the directories listed in the PATH variable:

$ ls /usr/bin

[ mysqlanalyze

2to3 mysqlbinlog

2to3-2.7 mysqlbug

a2p mysqlcheck

aclocal mysql\_client\_test

aclocal-1.11 mysql\_convert\_table\_format

...

mysql zdump

mysqlaccess zsoelim

mysqladmin

Depending on your computer, different types of files may have different colors. On many computers, for example, executables will probably be colored green. If you look at the files located in your home directory, however, they are probably white and blue, which tells us that they are not executable:

One of the items in the /usr/bin directory is man. If I type man on the command line and hit enter, it will execute that file. How can I be sure that it will execute that file, and not some other file that happens to be named man on my server?

When you type a word into the command line, and it doesn't start with a /, ~, or a . (because those would indicate a path to an actual file), the command line will search each of the directories listed in the PATH environment variable for that command. So when we type man and hit enter, the CLI searches /usr/local/sbin, then /usr/local/bin since those are first in the list. It then moves on through the list of directories until it finds a file named "man" in one of them. It then stops searching and executes the file. We can verify which file is getting executed by using the which command:

$ which man

/usr/bin/man

As you can see, the path that is output by the which command corresponds with the man file we found earlier.

You can create or install executables. To make it so that a custom executable can be used like a built-in command, all you have to do is make sure it has the correct permissions (discussed in the next chapter), and add the path to the directory it is contained in to the PATH variable in ~/.bashrc:

PATH="/path/to/my/executables-directory:$PATH"

Note how we added the executables directory, then the colon, then the $PATH variable again. This preserves the old PATH locations while making your directory the highest priority. We're adding our directory to the top of the list.

Note that if this path you're adding to the $PATH environment variable contains an executable file called "man", then when you type man from the command line, this new file will be executed instead of the correct one in /usr/bin. The path lookup rules for all commands relies heavily on managing this $PATH environment variable carefully.

To sum up:

* The PATH variable determines which directories are searched when a command is entered
* PATH is an ordered, colon-delimited, list of directories that contain executables
* The order of the directories in the PATH variable is first-found-first-execute
* If you use a /, ., or ~ before your command, the command line will interpret that as an actual path to a file, and will not use the PATH variable
* You can add to PATH to make more commands available without having to memorize their exact path
* Modifications to PATH, or any environment variable, on the fly will not be permanent; permanent modifications should be done in an environment file, like .bashrc

**Exercises**

1. Run the following commands to experiment with altering your command line environment:
2. $ cd
3. $ PS1="\u@\w$ "
4. ubuntu@~$ echo "Hello world"
5. Hello world
6. ubuntu@~$

Exit out of Terminal (make sure to close each tab and window if you are on a Mac) and open it up again. What does your prompt look like now? The value you set PS1 to above should no longer be in effect.

Set your prompt in your ~/.bashrc file:

$ echo 'export PS1="this is a test$ "' >> ~/.bashrc

The redirection operator (>>) is used to *append* text to a file. If the target file doesn't exist, then it will be created.

Exit and open up a new terminal. You should see something like this:

this is a test$

Edit the ~/.bashrc file, remove the last line, and run source ~/.bashrc to return your prompt to its previous state.

1. As discussed previously in this chapter, the PATH variable determines which files can be executed without specifying their path explicitly. Run through the commands in this exercise to see this principle in action.

Set the current PATH variable to another variable so we can revert to it later:

$ OLDPATH=$PATH

$ echo $OLDPATH

/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/opt/aws/bin:/home/ubuntu/bin

Temporarily change PATH and try to run a program that is no longer located in the directories specified in PATH:

$ cd /usr/local/init

$ ls

redhat ubuntu

$ PATH="" ls

-bash: ls: No such file or directory

$ ls

redhat ubuntu

Now, create an executable in your home directory by entering the following:

$ echo '#!/bin/bash' > ~/test.sh

$ echo 'echo "Hello world"' >> ~/test.sh

$ chmod +x ~/test.sh # Make sure test.sh has the executable permission

Run the executable, first by specifying the path, then by running it like a command:

$ ./test.sh

Hello world

$ test.sh

-bash: test.sh: command not found

Now change PATH more permanently for your current session, and try to run test.sh again:

$ PATH=$PATH:$HOME

$ echo $PATH

/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/opt/aws/bin:/home/ubuntu/bin:/home/ubuntu

$ test.sh

Hello world

$ cd /

$ test.sh

Hello world

Exit out of Terminal and then open it up again. Try running test.sh again:

$ test.sh

-bash: test.sh: command not found

Permissions

**[Overview](https://launchschool.com/books/command_line/read/permissions" \l "overview)**

A huge component of the command line environment that we've mostly ignored up until this point is your user account. When you open up Terminal or otherwise connect to a server, you are logging in to the command line. Your user account determines which directory you are put in when you connect, and it also determines what permissions and access you have to files and directories. Everything we've talked about up to this point is files, directories, executables, and the environment. Permissions are no different; they are based in the file system and depend on the user account for context.

**Note:** Permissions can be confusing. Don't get discouraged, though! A basic understanding of permissions can get you a long way. After reading through this chapter, make sure you do all the exercises to solidify the concepts discussed here.

In Unix and Linux file systems, permissions are divided into two parts: ownership and access types. There are three levels of ownership: **user**, **group**, and **other**; and three access types: **read**, **write**, and **execute**. Each level of ownership can be granted any or all of the access types, which brings the total number of permission combinations to 29, or 512. Luckily, it's very easy to see what permissions a file or directory has, and also pretty easy to modify its permissions.

**[Interpreting Permissions](https://launchschool.com/books/command_line/read/permissions" \l "interpretingpermissions)**

In order to understand permissions better, let's think about a VIP lounge at the airport. Many people have access to the VIP lounge: airport staff, custodians and, of course, VIP members. The lounge has an owner—Acme Airline, for example—that has full access and control. There's an implied permission level for the general public as well (they cannot enter the VIP lounge). Once inside the lounge, there may be more access points that have different permission levels. The custodian's closet, for example, will be locked so that only custodians have access to it. Custodians, in turn, probably aren't supposed to drink the "free" beverages or otherwise use items reserved for the VIP members.

Let's represent the VIP lounge in terms of a file structure (note that tree is a command that displays a directory and its sub-directories):

$ tree airport/

airport/

└── vip\_lounge

├── coffee

├── custodian\_closet

├── donut

└── internet\_access

If we were to run ls with flags -lah on that file structure, we'd probably get something like the following (Remember that .. means "the parent directory," so in this case it refers to the airport/ directory.):

$ ls -lah vip\_lounge

total 0

drwxr-xr-x 6 acmeair vip 204B Jul 14 15:24 .

drwxr-xr-x 7 acmeair public 238B Jul 14 15:23 ..

-rw-r--r-- 1 acmeair vip 0B Jul 14 15:23 coffee

drwxrwx--- 2 acmeair acmeair 68B Jul 14 15:23 custodian\_closet

-rw-r--r-- 1 acmeair vip 2B Jul 14 15:24 donut

-rwxr--r-- 1 acmeair vip 1GB Jul 14 15:24 internet\_access

As mentioned above, permissions have three access types: read (r), write (w), and execute (x). Both files and directories have access types assigned to them, and each can be assigned a user and a group. The "other" level of ownership always includes everyone else implicitly. It looks like this:

* A file or directory has:
  + an owner + owner's rwx access levels
  + a group + group's rwx access levels
  + other's rwx access levels

In the vip\_lounge example above, you can see to the left of each file a string of characters that starts with a dash (which can also be a d to indicate a directory), then 9 more characters consisting of r ("read"), w ("write and delete"), x ("execute"), and - ("permission not granted"). Where a letter exists, that access type is granted, and where the dash exists, that permission is not granted.

After the string of d, rs, ws, xs, and dashes in the first item in the list, you can see acmeair vip. These two strings represent the user and group assigned to the file or directory. In this case, the name of the user is "acmeair" and the name of the group is "vip".

The following chart shows what each part of the permissions represents. Each of the first 10 characters can be replaced with a dash to show that access level is not granted or, in the case of the d, that the object in question is not a directory. Let's take a look at a diagram to make this a bit easier to understand:

# +-------- Directory or not

# | +------- User Read, Write, Execute

# | | +------- Group Read, Write, Execute

# | | | +----- Other Read, Write, Execute

# | | | | +--- The name of the user

# | | | | | +--- The name of the group

# | | | | | |

# d|rwx|rwx|rwx user group

Now that we understand how the permissions are laid out, we can interpret the output of the ls -lah command we ran previously. If you look at the first item displayed, the . directory (also vip\_lounge), we can gather the following:

* d: It is a directory.
* rwx: Its user has read, write, and execute access.
* r-x: Its group has read and execute access.
* r-x: Other also has read and execute access.
* The user named "acmeair" has been assigned to it.
* The group called "vip" has been assigned to it.

In practice, this means the directory's default permissions allow for files to be read and executed by any user, and written to by the user "acmeair."

In this hypothetical file system, the vip group would have multiple users associated with it, such as acmeair, bob, dave, etc. Let's pretend that the following users exist and belong to the groups listed beneath them (we'll be talking about groups more in a later section):

$ groups bob

bob vip

$ groups acmeair

acmeair vip

$ groups jeff

jeff acmeair

In this scenario, anyone that belongs to the vip group would have group access to anything that is assigned the vip group. The user bob, based on his groups as listed above, would have the ability to enter the vip\_lounge directory and partake of (i.e. read) its donut, coffee, and internet access content. The bob user, however, wouldn't have access to enter the custodian\_closet content or use its contents, but jeff and acmeair would.

An important thing to note from the above example is that the permissions are *not* bestowed upon the user or groups. Rather, the files and directories are assigned groups and users, with particular permission levels for those. It's kind of like a reservation at a restaurant table. There's nothing about a person that inherently gives them permission to dine at a restaurant at a particular table and at a particular time. It's because the restaurant has labeled, or set apart, that table for that time that the person is able to dine there. So it is with the file system. There's nothing inherently special about any group or user (except the root user). A user's privileges are defined by the files and directories themselves.

**[Setting Permissions](https://launchschool.com/books/command_line/read/permissions" \l "settingpermissions)**

There are a few ways of setting the permissions on a file or directory. The chmod command will be your friend here. To add write permissions to a file, for example, you can do something like this:

$ chmod +w sample.txt

The +w means "add write access." If you want to get more granular in how you set permissions on a file or directory, you can prefix the permission with u, g, o, or a, which stand for "user (owner)", "group", "other", and "all", respectively:

$ ls -lah test.txt

-rwxr--r-- 1 bob staff 1GB Jul 14 15:24 test.txt

$ # Remove write access for user

$ chmod u-w test.txt

$ ls -lah test.txt

-r-xr--r-- 1 bob staff 1GB Jul 14 15:24 test.txt

$ # Add execute access for group

$ chmod g+x test.txt

$ ls -lah test.txt

-r-xr-xr-- 1 bob staff 1GB Jul 14 15:24 test.txt

What if you want to set access level permissions for the user, group, and other all at once? You can do so with 3 numbers, each from 0 to 7. Why 0-7? There are 23 (which is 2 \* 2 \* 2, or 8) permission levels. When you count from 0 to 7, there are actually eight numbers. Note that it's really common in computer languages to count starting from zero rather than 1. In this octal system, execute, write, and read permissions each add 1, 2, and 4 respectively, resulting in non-ambiguous designations of permissions. In short, here are the different combinations of permissions based on the octal mask.

The following table shows what access level each number represents:

| **Number** | **Permission** |
| --- | --- |
| 0 | No permission granted. |
| 1 | Can execute. |
| 2 | Can write. |
| 3 | Can write and execute (2 + 1 = 3). |
| 4 | Can read. |
| 5 | Can read and execute (4 +1 = 5). |
| 6 | Can read and write (4 + 2 = 6). |
| 7 | Can read and write and execute (4 + 2 + 1 = 7). |

If you combine the permissions from the table above—one each for owner, group, and other—you can define the whole set of permissions for a file or directory:

$ chmod 777 test.sh

$ ls -l test.sh

-rwxrwxrwx 1 bob admin 0B Jul 15 15:24 test.sh

$ chmod 000 test.sh

$ ls -l test.sh

---------- 1 bob admin 0B Jul 15 15:24 test.sh

$ chmod 754 test.sh

$ ls -l test.sh

-rwxr-xr-- 1 bob admin 0B Jul 15 15:24 test.sh

**Note:** in order to change the permissions of a file or directory, you must be its owner, be root, or use sudo. See the [Root User and Sudo section below](https://launchschool.com/books/command_line/read/permissions#rootuserandsudo).

**[Users and Groups](https://launchschool.com/books/command_line/read/permissions" \l "usersandgroups)**

We know that a user and a group are assigned to all files and directories in Linux and Unix systems, and it is pretty obvious that if your user is assigned to a file, you will have the ability to read, write, or execute the file according to the access types granted. But how do groups work? Users can belong to multiple groups, and groups can have multiple users. If a user belongs to a group, it will have the access types granted to the assigned group of any file or directory. So, in our ls -lah example above, the file named "donut.txt" can be read (or consumed!) by any user in the "vip" group.

To determine if your user is part of a certain group, use the groups command:

$ groups

ubuntu adm dialout cdrom floppy sudo audio dip video plugdev netdev

In the above example, the "ubuntu" user is part of several groups, the first being a group of the same name as the user: "ubuntu".

**[Root User and Sudo](https://launchschool.com/books/command_line/read/permissions" \l "rootuserandsudo)**

In Unix and Linux systems, there is a special user called "root." The root user is the super user—it can read, write, and delete any file. If the "acmeair" user is like an airline company, then the root user is basically a god. Because the root user has so much power, it is a common recommendation to not log in as root, and to only run commands with root when necessary. If you are logged in as a non-root user and know the root user's password, you can switch to the root user account at any time with the following command:

$ su -

Password:

If you put in the above command, you will be required to put in the root user's password. Note: you may not have root access if you are using a managed server environment.

Although the root user can read, write, and delete (almost) any file, it cannot execute just any file. As mentioned in the chapter "Files, Folders, and Executables", a file can only be executed if it has the execute permission granted. In the case of the root user, it doesn't matter who the permission is granted to; as long as it is granted to the user, the group, or other, root can execute it.

Sudo

While you will be able to log in as root directly in some scenarios, it is more common to use the sudo command to perform actions that require root privileges. You can think of it as borrowing the god-like powers of the root user for a moment. The sudo command allows you to "do" something as a "super user." When you use this command, you will usually be required to input a password; but instead of the root user's password, you'll be putting in your own password. The following is an example of a command that requires root privileges, but that can be run using sudo (don't run this unless you want your computer to reboot!):

sudo reboot

But why use sudo instead of just logging in as root? There are several reasons for doing this, including the following:

* The server administrator wants you to have root access for some commands and/or directories, but not for everything. In this case the administrator will set up sudo to have restrictions or whitelisted commands.
* Running commands while logged in as root can be dangerous. Using a non-root user makes it obvious when you are running a command that requires root privileges because you have to prefix your command with sudo.
* The sudo command provides a detailed audit trail so that system administrators can track what commands individuals used on system files.
* Sudo uses a ticketing system where you put in your password once, then you don't have to until you haven't run any sudo commands for five minutes or longer. This adds security to your command line session, preventing others from gaining root access if you leave your Terminal open on accident.

**[Summary](https://launchschool.com/books/command_line/read/permissions" \l "summary)**

File system permissions are complicated, and are probably one of the biggest sources of headaches for command line beginners. But having a basic understanding of how permissions work will get you a long way. Remember:

* Permissions are assigned to files and directories—not users and groups.
* Access levels are determined by the rwx (read, write, execute) permissions for the owner, group, and other.
* A file must have the x (execute) permission to execute that file directly.
* To change the permissions of a file or directory, you either must be logged in as that file's owner or the root user, or you must use the sudo command.

With an understanding of the above, you'll be able to resolve many issues that you will surely run into while programming.

**Exercises**

1. Which user and group are assigned to the /etc folder on your computer?

**Solution**

1. Which user and group are assigned to the $HOME folder?

**Solution**

1. What are bob's permissions for the napkins file in the following outputs?
2. $ groups bob
3. bob travelers vip
4. $ ls -l custodian\_closet
5. total 0
6. -rw-rw---- 1 acmeair acmeair 0 Jul 21 17:57 napkins
7. -rw-rw---- 1 acmeair acmeair 0 Jul 21 17:57 paper\_towels

**Solution**

1. What are bob's permissions to the donuts file in the following example:
2. $ groups
3. bob travelers vip
4. $ ls -l vip\_lounge
5. $ ls -l ./
6. ...
7. -rw-rw---- 1 acmeair vip 0 Jul 21 17:57 donuts
8. ...

**Solution**

1. What are bob's permissions to the laptop file in the following example? What are the permissions for the group acmeinc?
2. $ ls -l vip\_lounge
3. ...
4. -r-xrwx--- 1 bob acmeinc 0 Jul 21 17:57 laptop
5. ...

**Solution**

1. Practice using sudo to log in as root:
2. $ cd
3. $ sudo su
4. Password:
5. root@host:/home/ubuntu#

Exit out of the root user session:

root@host:/home/ubuntu# exit

$

Log in as root in root's home directory:

$ sudo su -

Password:

root@host:~# exit

$

Run a command as root:

$ ls /root

ls: cannot open directory /root: Permission denied

$ sudo ls /root

Password:

$

**Note:** You may not get the same results on your computer when running the above command. Specifically, if you are on a Mac, you probably don't have a /root directory. The above is an example of what you may see in a typical Linux environment.

**Warning:** The root user, and by extension the sudo command, can be very powerful. As such, be very careful when you use it, especially with commands like rm, as you could potentially delete your entire file system by accident.

Other Interfaces

**[Changing context in Terminal](https://launchschool.com/books/command_line/read/other_interfaces" \l "context)**

We've talked a lot about how the command line is an interface to the files and directories on your computer, and how your current directory and environment variables provide context for what you do on the command line. There are some commands that you'll use on the command line that will completely change the context of the command line. In fact, these commands provide entirely new tools that can help you to interface more directly with files, to query databases, to experiment with scripting languages such as Ruby, JavaScript, and Python, or to otherwise manage your system. The following are the types of commands that change the context of the command line and provide new ways to interface with your computer:

* Database management commands: mysql, psql, redis-client, mongo
* Text editors: vim, pico, nano, emacs
* REPLs (Read-Eval-Print-Loop), which are basically interactive scripting consoles: irb, python, php -a
* System monitoring: top, htop
* Reading files or manuals: man, less, more
* Window/Session handling: byobu, screen, tmux

Each of the commands above provide an entirely new interface. Once you enter that interface, though, you'll have to exit out of it (usually by typing exit) to get back to the normal command line.

One very useful command that takes over the CLI display is top. When you run top, it takes over all of the available screen real estate and refreshes every 3 seconds. The top program uses single letter commands to adjust its output. Try running top, then press b, then z.

$ top

The display should look something like this:



Type q to exit top.

The mysql command, which is used to connect to and interface with a MySQL server, also takes over the command line interface. As you can see in the example below, once you have connected to a MySQL server, you'll see a new prompt (this will only work if you have mysql installed):

$ mysql -u root -p

Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 4

Server version: 5.6.13 Source distribution

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its

affiliates. Other names may be trademarks of their respective

owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>

The prompt is now simply mysql>, and commands such as ls, echo, and cd no longer work. Instead, only SQL and other MySQL commands are available for use. The mysql program essentially hijacks the command line session until you exit out using the QUIT command.

Connecting to a MySQL server with the mysql command allows you to query databases and otherwise manage the database server. Just like the regular command line interface, there is input (STDIN) and output (STDOUT). If you type SHOW DATABASES;, it will print to the screen a list of databases on the server. If you type SELECT 1 + 1;, it will output a table with the result of 1 + 1.

To get back to the command line interface, type 'QUIT' and hit enter.

**[Read-Eval-Print-Loop (REPL)](https://launchschool.com/books/command_line/read/other_interfaces" \l "repl)**

Programs that follow the pattern of reading user input, evaluating the input, printing results of the input to the screen, and then allowing for more input, are called "REPLs", or "Read-Eval-Print-Loop" programs or interfaces. The mysql command provides a simple REPL. The bash shell, which is another name for the command line interface that we've discussed in this book, is also a REPL. Interactive interfaces for scripting languages such as Ruby, JavaScript, Python, and Perl are commonly referred to as REPLs—even more so than database clients and command line shells. These REPLs read user input, evaluate it using the scripting language's interpreter, print the result of the programming statement, then provide a prompt for more input.

Ruby REPL (irb)

If you don't have Ruby on your system, the commands in this section won't work. That's okay: you don't need Ruby right now.

In your console, try using the irb REPL. If Ruby is installed, you should see something like this:

$ irb

irb(main):001:0>

Just as with the mysql program, irb provides a prompt. As you can see in the following image, the irb program evaluates input, then performs what the input tells it to do, then outputs the result of what it did:

$ irb

irb(main):001:0> a = 1

=> 1

irb(main):002:0> puts "Hello, world!"

Hello, world!

=> nil

irb(main):003:0>

If you type a multi-line script, it will automatically postpone evaluating the input until the statement is finished:

$ irb

irb(main):001:0> a = 1

=> 1

irb(main):002:0> if a == 2

irb(main):003:1> puts "A is 2"

irb(main):004:1> else

irb(main):005:1\* puts "A is not 2"

irb(main):006:1> end

A is not 2

=> nil

irb(main):007:0>

Type exit to leave irb.

JavaScript REPL (node)

If you don't have Node on your system, the commands in this section won't work. That's okay: you don't need Node right now.

In your console, try using the node REPL. If node is installed, you should see something like this:

$ node

Welcome to Node.js v12.10.0.

Type ".help" for more information.

>

As with the mysql program, node provides a prompt. As you can see in the following code, the node program evaluates input, then performs what the input tells it to do, then outputs the result of what it did:

$ node

Welcome to Node.js v12.10.0.

Type ".help" for more information.

> a = 1

1

> console.log("Hello, world!")

Hello, world!

undefined

>

If you type a multi-line piece of code, it will automatically postpone evaluating the input until the statement is finished:

$ node

> a = 1

1

> if (a === 2) {

... console.log("A is 2");

... } else {

... console.log("A is not 2");

... }

A is not 2

undefined

>

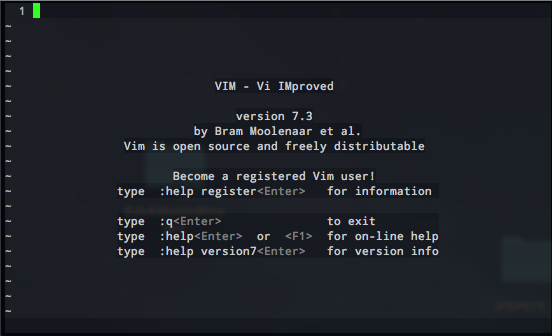
Press Control-D or Control-C to leave node.

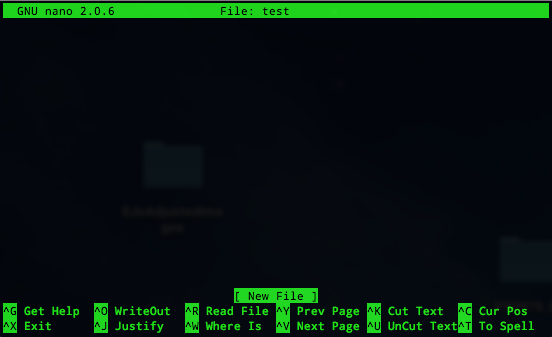
Other Scripting Languages

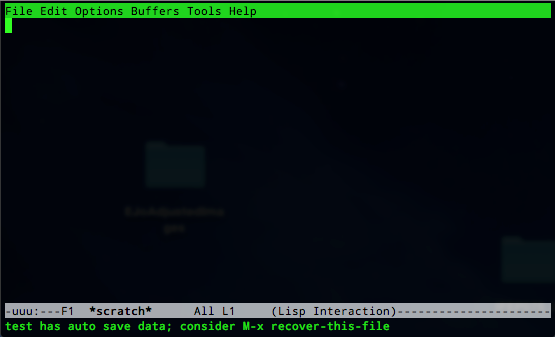
Other scripting languages, such as Haskell, Python, and PHP, also provide REPLs. As you learn to program, you will find that REPLs are very useful for experimenting with what you learn and for performing small computational tasks.

**[Editors](https://launchschool.com/books/command_line/read/other_interfaces" \l "editors)**

Some programmers use the command line as their IDE (Integrated Development Environment). Part of that IDE is the text editor, of which there are several available for the command line. The following interfaces may look familiar:







The above screenshots are all of text editors that are often used on the command line. When you see other developers working in interfaces like the ones above, they are probably not in the main shell REPL (the command line), and are instead using a text editor program within the command line.

**Note:** If you start typing command line commands while in an editor, those commands may be treated as normal text, or may be treated as editor commands. Watch out for that! You must exit the editor mode, and be back in the command line to issue command line (or "bash") commands.

The process of exiting is different for each of the editors mentioned above. Below is a list of the commands to exit each editor:

Vim

Exit and save:

<ESC> + :wq

Exit without saving:

<ESC> + :q!

Nano

Exit and save:

<Ctrl> + o then <Enter>, then <Ctrl> + x

Exit without saving:

<Ctrl> + x then n to discard changes

Emacs

Exit and save:

<Ctrl> + x, <Ctrl> + s, then <Ctrl> + x, <Ctrl> + c

Exit without saving:

<Ctrl> + x, <Ctrl> + c, then n, <Enter> and yes, <Enter> to discard any changes.

**[Bash: a Special Interface](https://launchschool.com/books/command_line/read/other_interfaces" \l "bash-interface)**

Sometimes, you may end up in a situation where you're working in a "command line within a command line". This is because the command line interface we've been working in is itself a special interface that can be nested. For example, while using the command line you may think that you're in the normal command line interface, but when you type exit and hit enter, you remain in what appears to be another CLI. This is most likely because you've unknowingly entered another bash (or shell) interface. When on the command line, typing bash, zsh, or similar will create a new nested shell interface for you to work in. You can exit the innermost "shell" by typing exit. When you're in the outermost shell and you type exit, you'll see something like [Process completed] without a way to enter more commands. You'll have to close and restart Terminal at this point and start a new shell. If you're working normally, you should very rarely encounter a nested command line situation, but it can happen.

[**Summary**](https://launchschool.com/books/command_line/read/other_interfaces#summary)

When you open up Terminal and start using it, you may not always be using the default command line: bash. Programs like mysql, vim, and irb provide interfaces of their own right within Terminal. It's very important to understand whether you're in an application-specific REPL, like mysql, or if you're in the bash shell REPL (aka, the command line). Sometimes, the only clue you have is the prompt. But as we've seen earlier, we can even modify the prompt by setting the PS1 environment variable, so it can be quite tricky.

Conclusion

[**Overview of Concepts**](https://launchschool.com/books/command_line/read/conclusion#overview)

Hopefully by this point, the command line is a bit less intimidating. While we've covered the basic concepts of the command line in this book, there's a lot more to learn. But don't get discouraged! You have all the tools you need to become an adept CLI user: this book, the man command to get more information about commands, and your favorite search engine.

Let's briefly review the concepts we discussed in this book.

· The command line is an interface to your computer's files and directories.

The command line provides a way to communicate with your computer. Its language takes the form of text-based input and output.

· Everything you do in the command line is related to files, directories, and executables.

When working in the command line, everything is a file, a directory, or an executable (and executables are files). When you run commands such as cd or ls, you are really just executing files.

· Environment variables provide context for what you do in the command line.

Context for the commands you run is provided by environment variables. Your "current directory" (PWD), the list of directories that defines the location of executables (PATH), and your home directory (HOME) are examples of environment variables that affect the commands you run and your ability to run commands. You can change environment variables on the fly using the VARIABLE="value" pattern, or you can set them more permanently by exporting the variables in your ~/.bashrc file.

· Access to files, directories, and executables is determined by their read, write, and execute permissions. There are permissions for the file's user (owner), group, and other (everyone else).

Another important piece of context is the user account you are currently logged in under. Each directory and each file is assigned to a user and a group, and permissions for reading, writing, and executing are granted to the user and group that are assigned to that directory or file. Permissions are also granted to everyone else ("other"), which includes any user that is not the user or doesn't belong to the file's group.

· Some programs provide a completely new context within the command line interface.

While the environment variables and your user account provide context for commands that you run while in the command line interface, some commands can temporarily take over handling all input and output. These commands allow users to interact more directly with databases, files, and scripts. They change the display and accept commands that aren't available in the regular command line interface, and commands that are available in the regular CLI aren't available in these other interfaces. Some of these interfaces are called "REPLs" because they **R**ead input, **E**valuate the input, **P**rint the results, and **L**oop back to read more input.

**[Where to Go from Here](https://launchschool.com/books/command_line/read/conclusion" \l "wheretogofromhere)**

With the basic command line skills you learned in this book, you are prepared to begin learning to program. Continue to practice what you've learned and experiment with the command line so that you can retain and improve on what you've learned. The more you use the command line, the easier it will get, and the more powerful you will become in your ability to make your computer do your bidding.