Learn a Command-line Interface

A modern introduction to age-old sorcery.

Kris Jordan

Contents

Why a command-line interface? **i**
0.1 Special Thanks i

Getting Started **iii**
0.2 Windows Installation iii
0.2.1 Update to Windows 10, Version 2004+ iii
0.2.2 Installing WSL2 iv
0.2.3 Installing Docker iv
0.2.4 Installing Windows Terminal iv
0.2.5 Installing `git` and Cloning the `learncli` Repository iv
0.2.6 Beginning a Shell Session v
0.3 macOS Installation v
0.3.1 Update to Catalina 10.15.6+ v
0.3.2 Installing Docker v
0.3.3 Installing `git` and Cloning the `learncli` Repository v
0.3.4 Beginning a Shell Session vi

1 The Sorcerer's Shell **1**
1.1 Your First Spells 2
1.2 Learn about a program with its `--help` argument 3
1.3 Paginate output with the `less` program 3
1.4 Read program manuals with the `man` program 4
1.5 Print file contents with the `cat` program 5
1.6 Autocomplete commands with the `tab` key 5
1.7 Reuse previous commands with the `up`/`down` keys 6
1.8 Clear your terminal screen with the `clear` program 6
1.9 Search and filter text with the `grep` program 6
1.10 Review your command log with `history` 9
1.11 End a shell session with the builtin `exit` command 9
1.12 Command Reference 9
1.13 Keyboard shortcuts in `less` 10

2 Directories, Files, and Paths **11**
2.1 Directories 11

2.2 Absolute Paths 12
2.3 Print your working directory with the `pwd` program 13
2.4 Change your working directory with the `cd` builtin 14
2.5 Relative paths from your working directory 14
2.6 The `/mnt/learncli` directory is shared with your PC 15
2.7 Create directories with the `mkdir` program 16
2.8 Copy files with the `cp` program 17
2.9 Silent success, noisy errors, and `--verbose` mode 17
2.10 Hidden "dot files" begin with a `.` 18
2.11 Long vs short options and case sensitivity 18
2.12 List hidden files with `ls --all` or `ls -a` 19
2.13 Parent directory `..` and current directory `.` links 19
2.14 Move or "rename" files with the `mv` program 21
2.15 List directories recursively with the `find` program 22
2.16 Delete files with the `rm` program 22
2.17 Delete directories with `rmdir` or recursive `rm` 23
2.17.1 Delete empty directories with the `rmdir` program 23
2.17.2 Delete non-empty directories with `rm -i --recursive` 23
2.18 Delete files and directories with *caution* 24
2.19 Command Reference 25

Why a command-line interface?

Computers are machines. Their original intent was to help us do work at previously unthinkable scales analyzing large data sets, generating reports, finding patterns, and so on. As the availability and distribution of computers spread beyond universities and businesses and into our backpacks and pockets, their widespread use is as much about *accessing* and *sharing* information as it is about *doing work*. In this transition, most people interacting with computers are typically software *consumers* as opposed to system *operators* and software *creators*.

If you want to succeed as a *data-driven manager*, an *analyst*, an *engineer*, or a *scientist*, the ability to effectively *operate* a computer and *develop* programs will give you super powers. The command-line interface popularized by Bell Labs' Unix system in the 1970s continues to serve as the foundation of modern operations and development. Despite the advent of operating systems with graphical user interfaces such as Microsoft's Windows or Apples macOS, the textual interface of a command-line offers much more power, flexibility, and simpler automation if you're willing to climb your way up its steeper learning curve.

The purpose of this text is to help you gain comfort working at a command-line. It assumes no prior experience and intends to help you establish a mental model of the shell, file system, process model, and important tools that is technically sound without being pedantically overwhelming. After you understand a handful of key concepts, a new world of possibilities and capabilities opens up to you.

0.1 Special Thanks

Many colleagues and students provided feedback and corrections to this text. Special thanks to Madison Huber, Helen Qin, Jeffrey Young, Solomon Duncan, Christina Barta, Baron Northrup, and Eric Schneider for their suggestions and corrections.

Getting Started

The examples in these lessons center around a Unix-based operating system. Unix's ideas date back to the early 1970s. These ideas have withstood the test of time.

A historical challenge of learning these ideas and skills was gaining access to a system configured for learning. No more!

Thanks to innovative, free, open source software, you can gain access to the same command-line tool bench used by world class engineers and data scientists, on your personal computer. The learning exercises in these lessons give you a *sandbox*: the programs you run are isolated from your primary operating system and files so as to avoid any accidental deletion of important files. You are safe to, and *encouraged to* experiment!

The sandboxing you'll use in these tutorials are provided through containers. A container allows you to run programs in a controlled, reproducible environment. Containers were invented to simplify industrial systems operations and are used as infrastructural building blocks by Google, Amazon, Facebook, and more. In the past 10 years, container technology became more user friendly and readily accessible.

Containers offer educational benefits, too. Once containers are up and running on your personal computer you can quickly and easily replicate the systems, tools, and configurations used throughout these tutorials.

0.2 Windows Installation

0.2.1 Update to Windows 10, Version 2004+

Docker for Windows runs best on Windows Subsystem for Linux 2 (WSL2), a feature that is included in the lastest versions of Windows but requires some configuration to enable. Check your Windows version number before continuing:

Press `Windows Button + R`, type `winver`, and press Ok or Enter. You should see at least:

- Version 2004 or later
- Build 19041 or later

If it is not, visit the following URL and look for the Windows 10 May 2020 Update area. Follow the steps needed to upgrade your operating system:

https://www.microsoft.com/software-download/windows10

0.2.2 Installing WSL2

Microsoft's instructions for enabling WSL2 have steps that involve "Open PowerShell as Administrator". PowerShell is a Windows-specific command-line interface. Running it as "Administrator" gives it privileges to modify system settings it couldn't otherwise. To do this, you will need to open your start menu, search for PowerShell, rightclick on it and *Run as administrator*.

Follow Microsoft's instructions for enabling WSL2 by completing the only the first two sections of the following guide: https://docs.docker.com/docker-for-windows/wsl/ - you do not need to install a Linux distribution, or the steps after it.

Update the WSL2 kernel by following these instructions: https://docs.microsoft.com/en-us/windows/wsl/wsl2-kernel

Finally, run Powershell as an Administrator once more, and then run the following commands (press enter after each line):

```
wsl --set-default-version 2
Set-ExecutionPolicy RemoteSigned
```

0.2.3 Installing Docker

Follow Docker's instructions to install Docker Desktop Stable:

https://docs.docker.com/docker-for-windows/wsl/#download

You can stop at Step 4 after confirming Docker is using the WSL2 based engine.

0.2.4 Installing Windows Terminal

The recommended Terminal software for this course is the Windows Terminal app made by Microsoft. You can install it from the Microsoft Store app by searching for "Windows Terminal".

0.2.5 Installing `git` and Cloning the `learncli` Repository

After installing Windows Terminal, start it normally, *not* as an administrator.

Confirm whether you have `git` installed by running the command:

```
git --version
```

If you get an error, please follow the instructions to download and install git here:

https://git-scm.com/downloads

After `git` is installed, restart Windows Terminal normally and run the following `git` command:

```
git clone https://github.com/comp211/learncli211.git
```

You'll learn exactly what this command is doing, and much more, soon!

0.2.6 Beginning a Shell Session

After completing the install steps above, you should be able to navigate into the `learncli211` directory with the *change directory* (`cd`) command and then run a *PowerShell* script (the `.ps1` file) to start up the `learncli` container. These are the steps you'll follow in the future to get back into your `learncli` container:

```
cd learncli211
./learncli.ps1
```

If all is well, then you will see some UNC CS ASCII art and a `learncli$` prompt string awaiting your command. Woohoo! That's all you need to accomplish for now.

End your container session by running the command `exit` command, don't just close your Terminal Window else the container will run in the background.

0.3 macOS Installation

0.3.1 Update to Catalina 10.15.6+

Confirm your mac operating system is Catalina, version 10.15.6 or greater, by opening the Spotlight Search and searching for *About this Mac*. If your operating system version is not 10.15.6 or greater, click the *Software Update...* button and install recommended updates.

0.3.2 Installing Docker

To install Docker on macOS, please follow the official instructions:

https://docs.docker.com/docker-for-mac/install/

0.3.3 Installing `git` and Cloning the `learncli` Repository

Open a Terminal via *Applications* > *Utilities* > *Terminal*, or by searching for it in Spotlight Search.

Run the command:

```
git --version
```

If prompted that you need to install git, follow the instructions to install it. If you see an error, try running `xcode-select --install`

Once `git` is installed, close and open a new Terminal window and run the following command. You'll learn more about what exactly this is doing soon!

```
git clone https://github.com/comp211/learncli211.git
```

0.3.4 Beginning a Shell Session

If you do not have a Terminal window open, go ahead and do open one via *Applications* > *Utilities* > *Terminal*, or by searching for it in Spotlight Search.

Then, run the following:

```
cd learncli
./learncli.sh
```

If all is well, then you will see some UNC CS ASCII art and a `learncli$` prompt string awaiting your command. Woohoo! That's all you need to accomplish for now.

End your container session by running the command `exit` command, don't just close your Terminal Window else the container will run in the background.

Chapter 1

The Sorcerer's Shell

Imagine a wooden stick, a couple feet long, mostly straight, and tapered at one end. Boring, right?

Now imagine a sorcerer's wand in the hand of a student at a wizarding school on the brink of casting her first spell. *It's the same stick.*

Welcome to wizarding school!

The line below, which you *should* see as the last line in your terminal[1], is a *bash* command-line interface prompt or, more commonly, a *shell* prompt. The blinking cursor awaits your spell.

```
learncli$
```

The power resting beneath your fingertips is unbounded; before you is a workbench that for the last 50 years has been used to investigate and publish great research, control large-scale systems, and build software empires.

Looking at your terminal screen and reading these words you may be thinking something is amiss, "...that's it? Are we really looking at the same thing?"

Yes. Yes, we are.

How you imagine a tool's powers determines your perceptions of it. Approach learning the command-line with the same sense of awe and wonder as learning to wield a sorcerer's wand. At the very least you'll enjoy yourself more. Hopefully, though, you'll start to recognize you *are* on a journey into modern day wizardry.

[1] If you *do not* see the `learncli$` prompt, please return to the previous chapter, Getting Started, and complete the steps to install the software necessary for these tutorials.

1.1 Your First Spells

At first the shell will look intimidating, but don't worry! In this section you'll get a feel for working in a shell before diving into its details. The shell is your concierge to a system. Its purpose is to help you do work by running and operating programs as you wish.

A natural question you might ask is, "what programs can I run at a command-line?" Far more than you'd think! Many standard programs, often called *system utilities*, are available anytime you are working in a unix-like operating system. These programs are special kinds of files stored in specific directories. One such directory is the `/bin` directory. The word "bin" is short for "binary program files" and stores files which your computer can evaluate as a program[2]. You can *list* these files with the following `ls` command:

```
learncli$ ls /bin
bash           grep           ntfs-3g.probe  su
bunzip2        gunzip         ntfscat        sync
...
```

As a convention of these tutorials, what *you* type is preceded by the *prompt string* `learncli$`. Unless otherwise obvious, the commands you type will be lowercase characters with some spaces, special symbols, and numbers. When you press enter, as you did after typing `ls /bin`, the command-line interface reads your command, interprets it, and attempts to carry out your request.

Look at all of those programs! Don't stress, you only need to know a few of these to be productive. Most you'll *never* use, unless you specialize in systems administration. There are more utility programs on this system than most people have apps on their phones. Most command-line programs are intentionally designed to handle one specific kind of task.

Scan through the output of the command above on your terminal and be sure you find the file named `ls` in that list. Did you find it? Does `ls` ring a bell? It's the same two letters as the start of the command you wrote above. This isn't a coincidence, *they're the same thing*! To list the files in the `/bin` directory, you executed the `ls` program which is a standard utility program found in the `/bin` directory! The `ls` program correctly listed *itself* as a file in the `/bin` directory. Listing files in directories is the sole purpose of the `ls` program.

Unix program names are short, cryptic, and, at first, mysterious. There are a few ways to learn what a command-line program is useful for and how to use it. The most common, consistent advice is to run a program in `--help` mode or to "read the manual" (often abbreviated as RTM). This advice is not wrong, but what you are presented may contain terminology beyond your current level of comfort. My recommendation is to first attempt to read `help`, then scan the manual and, assuming the command still does not

[2] These days you will commonly find *scripts* in directories with `bin` in their name, too. A script is the source code of a program written in a scripting language like Bash, Python, JavaScript, or Ruby, and is a just a plain text file, not a binary file like most system utility programs are. Thinking of directories with `bin` in their name as storing "runnable programs", whether via a scripting language interpreter or as binary files directly executable at the machine level, is encouraged.

make sense, *then* search the internet for plain English explanations. This will help you learn new vocabulary quickly.

1.2 Learn about a program with its `--help` argument

Command-line programs usually have a `--help` argument that prints some information about the program's purpose, usage, and options. When you run a program in `--help` mode, the program itself doesn't attempt to carry out any task other than providing you with information. Try running `ls` with a `--help` argument.

```
learncli$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
...
```

There is too much text to digest for a quick, casual reading. The *list files* program, `ls`, has around 40 different optional arguments[3]. Not only are there a lot of specifically purposed command-line programs, each also has a range of options and capabilities. I point this out because a natural response is to feel overwhelmed, but I want to assure you the number of concepts you *need to know* is far less.

1.3 Paginate output with the `less` program

The `--help` text for `ls` is so long its basic usage and description likely scrolled off the top of your screen. To see *less* output, or, specifically a single screen of output at a time, use `less`.

Try running the following command:

```
learncli$ ls --help | less
```

Only one screen of information displayed. When using `less`, you can press the `f` button to scroll down a page at a time, the `b` button to scroll up a page at a time, and the `k` and `j` buttons to scroll up and down a line a time respectively. Press the `q` button to *quit* and return back to your shell's prompt.

Use `less` to paginate any program's output. After quitting the previous `less` program (described above), try using `less` in conjunction with the previous `ls /bin` command:

```
learncli$ ls /bin | less
```

As you scroll through the list of programs in the `/bin` directory, note the name of the first program whose name begins with an `l`. Yes, `less` is *just another program*, like `ls`. Again, you can exit `less` with `q`.

[3] The `ls` program dates back to the original Unix operating system in 1969, so it's 50 years old! For an often-used program to survive it naturally accumulates features and capabilities to aid its flexibility.

When you ran the commands using `less`, the vertical bar `|` character formed a connection, called a *pipe*, between the `ls` program on the left and the `less` program on the right. The pipe supported your first experience *composing programs*, an important Unix command-line concept. With pipes, it is easy to connect programs together, to mix and match the outputs of one program as the inputs of another, leading to a multiplicative effect on tasks you can carry out. In a simplified, contrived example, imagine having 10 programs capable of producing output data, and 10 programs that "filter", or process, data as input, then you have over 100 combined capabilities[4] from only 20 programs.

1.4 Read program manuals with the `man` program

The second way to learn more about a program is to read its manual. The program to read the manuals of other programs is the shortened prefix `man`. Try running the command below:

```
learncli$ man ls
```

Your terminal's content is replaced with the manual for the `ls` program. Manual pages have a consistent, improved formatting over `--help`. Notice you only see one page at a time by default. Behind the scenes, the `man` program uses `less` to display its information. Thus, you scroll the manual using the keys you used to navigate `less`.

Most manuals contain all of the information of the program's "Help" mode and more. Good manuals will accurately and succinctly describe a program and how to use it. Every standard program should have manual documentation you can access via `man` followed by the program name.

Try reading the name and first few sentences of description for the program we'll use next, `cat`.

```
learncli$ man cat
NAME
       cat - concatenate files and print on the standard output
```

As mentioned earlier, help text and manual pages can include terminology you may be unfamiliar with. The best way to make sense of unfamiliar terms is to search the internet. There is a wealth of reference information, tutorials, and example uses of command-line concepts found on-line. At the time of writing, a simple Google search for *what is the cat command useful for* produces 36 million results. Without even leaving the results page an easy to understand synopsis is shown: "`cat` reads data from the file and gives its content as output." Let's try out `cat`!

[4] You can, and occasionally will, create pipelines of *more than two* programs, so the upper limit on combinations is actually much larger.

1.5 Print file contents with the cat program

Preinstalled on your Learn CLI container is a "dictionary" file containing over 100,000 words. Whereas the `ls` program is found at the path `/bin/ls`, the dictionary file is found at the *path* `/usr/share/dict/words`. The next chapter focuses on files, directories, and paths. For now, try using `cat`, as shown below, to read the words file and print its output to your terminal:

```
cat /usr/share/dict/words
```

Depending on your computer, it may take a minute for `cat` to finish; isn't it impressive watching a hundred thousand words fly across your screen? The `cat` program simply reads through the file we told it to, from top to bottom, and prints each line out one after another.

1.6 Autocomplete commands with the tab key

The shell performs autocompletion when you are typing in a command and press the `Tab` key on your keyboard. The autocompletion is not like your phone's, instead it is context dependent and based on the partial command entered. It only offers valid completions. Developing muscle memory to press `Tab` as you enter a command both helps you type commands faster and increases your confidence in their correctness.

To get a feel for autocompletion of paths with `Tab`, closely follow these steps at the `learncli$` prompt:

1. Type `cat /u` and press `Tab`. *You should see the path autocompleted to* `/usr/`.
2. Type `s` and press `Tab` once. *Nothing happens.* Press `Tab` once again. *You should see there are three valid paths at this point beginning with an* `s`.
3. Type `h` and press `Tab` again. *You should see the path is now* `/usr/share/`.
4. Type `di` and press `Tab` twice. *There are two directories in* `/usr/share` *beginning with a* `di`.
5. Type `ct` and press `Tab` again. *The path completes to include a trailing slash.*
6. Type `w` and press `Tab` again. *The path completes to* `/usr/share/dict/words` *and a space was automatically added after.*

In this example, `Tab` was autocompleting a file path for you. There are four subtle autocompletion behaviors illustrated in what you just saw.

1. When there is a single, unambiguous completion for the next part of a path, pressing `Tab` will insert the characters immediately.
2. When there are multiple, ambiguous completions, pressing `Tab` once does nothing and pressing it again lists all of the possible matches.
3. When a directory name is autocompleted, the shell will automatically insert a trailing `/` so that you can begin typing the next part of the path.
4. When a filename is autocompleted, the shell will automatically insert a trailing space character so that you can begin typing the next argument to the command, if there is one.

In addition to file paths, the `Tab` key can also autocomplete program names, too. Try the following at an empty `learncli$` prompt:

1. Type `ca` and press `Tab` twice. _All programs installed in the container beginning with the letters "ca" are printed. Notice `cat` is one of them, and `cal` is another.
2. Type `l` and press `Enter`. *The* `cal` *program prints a textual calendar representation with today's date highlighted.*

There are other instances and programs which support `Tab` based autocompletion, but autocompleting paths and program names are the two use cases you'll use most frequently.

1.7 Reuse previous commands with the `up`/`down` keys

If you want to read the contents of the dictionary file at your own pace, then you should pipe the `cat` command into `less`. Typing out the entire `cat` command again, with that long path, is somewhat tedious. Modifying or adding onto an even longer command previously run even more so.

To avoid the nuisance of retyping earlier commands, the Bash shell allows you to press the Up and Down keys on your keyboard to move back and forth between previously used commands in your history. This is frequently useful when you want to tweak or extend a previous command.

Go ahead and try using the Up and Down keys to reuse the `cat /usr/share/dict/words` command. Do you remember how to use a pipe and `less` to view the words one screen at a time? If not, refer back to the previous section and continue once you've done so. As a general rule of thumb, if some command produces too much output and is safe to rerun, then pressing up, typing `| less`, and pressing enter is faster than trying to scroll your terminal window itself up to find the start of the command's output.

1.8 Clear your terminal screen with the `clear` program

The `clear` program clears your terminal screen and resets your prompt to the top of the terminal. This is useful when a previous command generates a lot of output and you'd like to "clear your head." When you are ready to move to your next task, `clear` your screen, stand up and stretch, and you'll return ready for the journey ahead.

1.9 Search and filter text with the `grep` program

The `grep` program uses textual patterns, formally called *regular expressions*, to search for textual matches. In this section you will encounter a few simple patterns. The little language of a regular expression pattern is much more powerful than shown here and discussed in more depth later on.

When you open the manual pages for `grep`, in the `SYNOPSIS` section you will notice a few example uses. The first is:

```
grep [OPTIONS] PATTERN [FILE...]
```

Two important example conventions are illustrated here. First, you'll notice two parts of the usage are surrounded in square brackets: `OPTIONS` and `FILE...`. The square brackets tell you they are *optional* arguments. You do not need to specify any `OPTIONS`, and in these examples you will not, nor do you need to specify anything for `FILE...`, though in the next example you will. The trailing `...` after `FILE` indicates you are able to specify multiple files one after the other, delimited by spaces, and `grep` will search each of the files listed.

```
learncli$ grep motion /usr/share/dict/words
commotion
commotion's
commotions
demotion
demotion's
demotions
emotion
...
```

In the example above, there are not any `OPTIONS`, the `PATTERN` is "motion", and the only `FILES...` argument is `/usr/share/dict/words`. Notice only the lines of the dictionary file containing the string of characters "motion" was printed. This is an exact match pattern, not too dissimilar from what you are able to do when you search within a web page or word document. The power of regular expressions becomes more evident as you make use of operators. One such example is the ^ character which anchors a pattern to the "start of a line".

```
learncli$ grep ^motion /usr/share/dict/words
motion
motion's
motioned
motioning
motionless
motions
```

By placing the ^ in the front of the pattern `grep` only matches lines beginning with the characters "motion". Conversely, the $ character anchors a pattern to the "end of a line".

```
learncli$ grep motion$ /usr/share/dict/words
commotion
demotion
emotion
locomotion
motion
promotion
```

Only words ending in "motion" are displayed. The last special character we'll demonstrate in this preview of `grep` is the `.` character which matches "any character". Let's

combine all three characters into a single pattern that searches for four letter words starting with a "g" and ending with a "p".

```
learncli$ grep ^g..p$ /usr/share/dict/words
gasp
glop
goop
grip
gulp
```

This ability to find matches in text is useful beyond searching the contents of a file like a dictionary. What if you wanted to search for file names matching a regular expression pattern in the output of `ls`? You could save those file names to a file and use `grep` as shown above, but there's a better way. Let's look at the usage example of `grep` once more:

```
grep [OPTIONS] PATTERN [FILE...]
```

What does it mean for `FILE...` to be optional? What does `grep` search if no file is given to it? It searches any input *piped* into it! Let's try it.

```
learncli$ ls /bin | grep ^g..p$
grep
gzip
```

There are two programs in the `/bin` directory that begin with a `g`, end with a `p`, with any two characters between them. One of them is the `grep` program itself. The other is `gzip`, a program for compressing data. In this example, the `ls` program listed all of the files in `/bin`, those lines of text were *piped into* `grep`, which in turn only printed lines matching the pattern argument specified.

Command-line programs that *filter* data, such as `grep`, tend to operate in one of two ways. They will *either* accept a list of files to process *or* will operate on data piped into them. The ability to provide a list of files is a convenience feature. The ability to process data piped in is essential and far more powerful. Remember, you already know a program capable of reading data from files in `cat`. A common scenario when looking at the contents of a file with `cat` is wanting to search for a specific word. As you gain comfort with the command-line, you will instinctively press up and pipe to `grep`.

```
learncli$ cat /usr/share/dict/words | grep ^g..p$
gasp
glop
goop
grip
gulp
```

As a more elaborate example, let's connect `cat`, `grep`, and `less` through a series of pipes to paginate all words in the dictionary containing "fun".

```
learncli$ cat /usr/share/dict/words | grep fun | less
```

1.10 Review your command log with `history`

Try typing the command `history`. What you will see is the trail of commands you previously ran. This listing shows the same list the `Up`/`Down` buttons draw from when reusing prior commands. The `history` command is useful to help recall some sequence of commands you tried before.

The `history` command is one of very few "built-in" shell commands, meaning it *is not* an external program like those you saw in the `/bin` directory. At this point, for your purposes, this distinction is mostly irrelevant. It is mentioned only to acknowledge a few commands are built-into the shell, but the vast majority, as you've seen in `/bin`, are programs defined outside of it.

Composing `history`'s log of your commands with `grep`'s ability to search for text is a useful combination. At some point you will ask yourself, "how did I run that command the other day?" Equipped with a knowledge of `history` and `grep` you can quickly get to the bottom of it. As a check for your understanding, try finding all of the commands you just ran which involved the `less` program[5]. How about all commands which involved the `grep` program[6]?

1.11 End a shell session with the builtin `exit` command

To end your `learncli` shell session, run the `exit` command[7]. This command causes the shell's process to exit and will return your terminal's control back to your host PC. From here you can either issue another `exit` command to your host's shell, subsequently ending it, or closing your terminal window.

As you progress through these lessons and want to resume your work following along with this book, refer to the instructions in the *Beginning a Shell Session* section of the *Getting Started* chapter.

1.12 Command Reference

Program or Builtin	Description	Standalone Usage
`ls`	List Files in Directory	`ls [PATH]`
`cat`	Concatenate or "Read" File	`cat FILE [FILES...]`
`less`	Terminal Paging Program	`less [FILE]`
`man`	Manual Pages	`man PROGRAM`
`clear`	Clear terminal screen	`clear`
`history`	Display history of commands	`history`
`grep`	Filter/"Search" by Regular Expressions	`grep PATTERN [FILES...]`
`exit`	End a shell session	`exit`

[5] `history | grep less`
[6] `history | grep grep`
[7] Like `history`, the `exit` command is also a built-in shell command.

1.13 Keyboard shortcuts in `less`

Key	Motion
`f`	Page down
`b`	Page up
`j`	Scroll down
`k`	Scroll up
`q`	Quit

Chapter 2

Directories, Files, and Paths

Projects in the real world are organized in files and directories. Research projects involve data files, analysis scripts, and generated reports and charts. Programming projects have source code, configuration files, and build scripts. Even games' graphics and sound assets, leaderboards, and the ability to restore progress from the last time you played, all rely on files and directories under the hood. A *file system* helps you keep projects organized independently from one another.

If you are comfortable with files and folders on your macOS or Windows PC, then you will notice direct similarities navigating a file system via the command line. Each is a different user interface to the same concepts. These differences incur trade-offs. A *graphical user interface* (GUI), such as Windows File Explorer or macOS's Finder, is easy to learn because it only requires comfort with pointing and clicking. Working with a file system via a *command-line interface* takes more effort to learn, but gives you more power. Using a CLI, you can easily *automate* repetitive file system tasks, such as renaming 1000s of files from one file naming convention to another, that would take hours or days to complete using a GUI. It's likely you've never *needed* this powerful of automation before, but as you grow professionally your projects' size tends to grow with you. This is also true in inverse, as the scale of work you manage grows, your career tends to grow with it. Investing time in learning how to automate work at the command-line interface now can accelerate and expand your career opportunities in the future.

2.1 Directories

Directories are the fundamental unit of organization in a file system. Every directory can contain other directories in a hierarchical relationship. Directories also contain files. One *root directory* has all other directories and files as its descendants.

The term *directory* is idiomatic when working at the command-line, though you're likely familiar with the term *folder* in macOS and Windows. The words *directory* and *folder* are synonymous. We will choose to embrace the term *directory* moving forward.

As you know, the program to list the contents of a directory is `ls`. Previously, you listed the contents of the `/bin` directory. It contained many of the standard system command-line program files. Try listing the contents of the *root directory* next:

```
learncli$ ls /
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
```

Notice the `bin` directory is contained in `/` (the forward slash is how you refer to the root directory). The file system in your `learncli` container[1] traces its roots back to the original Unix file systems of the 1970s (macOS's file system does, as well). The names of directories in the root directory, such as `bin`, `dev`, `etc`, `usr`, `var`, and so on, have been around since the early days of Unix[2]. The organizational purpose of these system directories is not worth concerning yourself over for now.

List the files in the top-level `/usr` directory. This directory contains "user installed" programs, source code, and libraries which are widely useful but not required by the operating system.

```
learncli$ ls /usr
bin  games  include  lib  local  sbin  share  src
```

The previous chapter used `grep` to explore the dictionary file stored at `/usr/share/dict/words`. Notice in the output of the last command, `ls /usr`, there was a directory named `share`. Also notice the start of the dictionary file began with `/usr/share`. This is no coincidence and the pattern continues. Try listing the contents of the `/usr/share` directory.

```
learncli$ ls /usr/share
... many, many directories, including dict ...
learncli$ ls /usr/share/dict
american-english words
```

2.2 Absolute Paths

A *path* is the textual "address" of a directory or file in the file system. When you want a program to operate on a specific directory or file you will need to give the program the path to it.

You just encountered a few paths. First, the root directory's path is `/`. The path to the top-level `usr` directory, contained within the root directory, was `/usr`. The path to the `share` directory within the `/usr` directory was `/usr/share`. Are you catching onto the pattern?

Paths which begin with a forward slash, referencing the root directory, are called *absolute paths*. An absolute path is followed by a sequence of directory names separated by

[1] The `learncli` container's *sandboxing* from your PC keeps separate the container's files from your PC's. You are safe to tinker in the container's files without fear of breaking anything.

[2] The files stored in each top-level directory evolved over the prior 50 years and aren't always consistent with their original intent. Just like a city that with 50 years of booming growth is beholden to past decisions made without clairvoyance, so it goes for many of the file system's organizational decisions.

slashes. The order of directory names is important and conveys their relationship. Each subsequent directory name is contained by, or a child of, the previous directory name. For example, `/usr/share` refers to the `share` directory contained in the `/usr` directory, whereas `/share/usr` does not exist because there is no `share` directory in `/`.

The last name in a path refers to either a directory or a file. This part of a path is important because it is the "target" of the path. Technically, it is called the *basename*. The basename is what the path is specifically referring to. Most programming languages have a standard library function named `basename` that extracts this string from a path. As you should come to expect, there's also a simple CLI program to do the same:

```
learncli$ basename /usr/share/dict/words
words
learncli$ basename /usr/share/dict
dict
learncli$ basename /usr/bin
bin
```

As the basename is the final destination, what comes before it is the "path" of directory names leading you to it. The idiomatic term for this slash delimited sequence of directory names is the *dirname* of a path. Try using the `dirname` program with a few paths. The result of `dirname` is another *path* to the parent directory of the input path.

```
learncli$ dirname /usr/share/dict/words
/usr/share/dict
learncli$ dirname /usr/share/dict
/usr/share
learncli$ dirname /usr/bin
/usr
```

The big idea of an *absolute path* is it fully identifies and addresses a resource in a file system. Starting from the root directory, the names of subsequent directories narrow in on the exact location the path is referencing. The directories up to and including the parent directory of a path form its `dirname`, while the specific name addressed by the path is its `basename`.

2.3 Print your working directory with the `pwd` program

Imagine working on a task involving many files in a single directory. Typing all their absolute paths quickly becomes painfully redundant. Fortunately, there's a better way. When you need to work on many files in a single directory, you can tell the shell it is your *working directory* and then write shorter and less redundant "relative paths" to files from it.

Your shell maintains a current working directory as part of its state. The `pwd` program prints the path of your working directory.

```
learncli$ pwd
/mnt/learncli/workdir
```

The output tells you the current working directory's path is `/mnt/learncli/workdir`. Within our `learncli` container this path has special properties we'll return to later.

2.4 Change your working directory with the `cd` builtin

For now, change your working directory to a path we're more familiar with using the `cd`, the acronym of *change directory*, builtin command.

```
learncli$ cd /usr/share/dict
learncli$ pwd
/usr/share/dict
```

The first command changed your shell's working directory to `/usr/share/dict`. The second `pwd` command was not required to change directories, but its output confirms your working directory changed. When you need to work with files in a directory, you should now begin by changing your working directory to it with `cd`.

2.5 Relative paths from your working directory

Previously, you used the `ls` command followed by an absolute path to list the contents of a directory. If you take a peek at the manual page of `ls`, you will see the following:

```
SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
      List information about the FILEs (the current directory by default).
```

There are two observations of note. First, notice both `OPTION`(s) and `FILE`(s) are *optional*, denoted by their surrounding square brackets. Second, the description tells you it will list information about the *current directory* by default. You can infer that if you run the `ls` program without any arguments it is the same as running `ls` with the current working directory as the `FILE` argument. Continuing from the `cd /usr/share/dict` command of the previous section, give it a shot!

```
learncli$ ls
american-english  words
```

The contents of your working directory are listed without providing any path at all. If you try running `ls /usr/share/dict`, you'll see the same output because your working directory currently *is* the absolute path `/usr/share/dict`.

Further, where you used `cat` to output the contents of `/usr/share/dict/words`, you can now simply use the relative path `words` to refer to the same file.

```
learncli$ cat words | less
```

In this example, `words` is a relative path. Since your current working directory is `/usr/share/dict`, the relative path `words` is the same as `/usr/share/dict/words`

which you previously typed out in full. A *relative path* specifies only what comes *after* the working directory and is not preceded by a slash. Relative paths work with subdirectories, as well.

```
learncli$ cd /usr/share
learncli$ ls dict
american-english  words
learncli$ cat dict/words | less
```

In this use of `ls`, the relative path `dict` is given. Since you just changed the working directory to `/usr/share`, the relative path was equivalent to the absolute path `/usr/share/dict`. For the same reason, when you ran the `cat` program with the relative path argument `dict/words`, you were referencing the absolute path `/usr/share/dict/words`.

Your current working directory only changes after a `cd` command completes. This command is different from most of the previous command-line programs you've used in that `cd` is *not* a program, it is a command "builtin" to the Bash shell, just like `history` and `exit` were. The Bash shell is just a program, too, and when you start your shell session in the `learncli` container, it begins the shell program which you type into. Part of that program's job is to keep track of your current working directory. This is important to note because all of the other programs you've encountered, including `ls`, `cat`, `grep`, `less`, and so on, are *just little programs*, programs *you* could implement on your own.

It is especially important to recognize you can use either kind of path, absolute or relative, anywhere a path is expected. Working at the command-line, **you can freely substitute absolute paths with relative paths and vice-versa**. Now that you are familiar with using the builtin `cd` command to change your working directory, you will find relative paths more convenient. Occasionally, you will find absolute paths preferable, such as when changing your shell session's working directory to a far off place in the file system.

2.6 The `/mnt/learncli` directory is shared with your PC

The `learncli` container's file system is separate from your host PC's. Changes you make to files in your container's file system will only persist until you end your session and revert to their original state the next time you begin a `learncli` shell session. The wonderful benefit to this container-based learning environment is if you do something unintended you can `exit`, begin the `learncli` container again, and you are back in action.

The `/mnt/learncli` directory, however, is different. It belongs to your host PC's file system and is "mounted into" the `learncli` container. All files and directories within it are accessible and modifiable from both the `learncli` container and your PC. Your work in this book will be within the `workdir` directory, the files outside it are configuration files for the container.

Let's take a look at the directory in the container first, and then confirm the files also exist on your host PC.

```
learncli$ cd /mnt/learncli
```

```
learncli$ ls
README  docker-compose.yml  learncli.ps1  learncli.sh  workdir
```

Now, open a second terminal window on your PC. In it, change your working directory to the host's `learncli` directory that you established in the *Installing Required Software* section. (Note we are using `host$` as the prompt string only as a convention and yours is expected to be different.)

```
host$ cd learncli
host$ ls
```

As expected, the same files and directories you saw in the `learncli` container are also on your host PC. Now try opening this directory in your host PC's file explorer or finder. The program for doing so differs between the Windows and macOS operating systems, so be sure to run the correct of the following commands (the trailing period or "dot" is important to both, so don't leave it off).

- Windows: `host$ explorer .`
- macOS: `host$ open .`

You should see a new window open with the current working directory on your host machine. Notice this is a *graphical user interface* view of the shared directory. In the upcoming section you'll create files in the `workdir` subdirectory shared between your host PC, and thus editable in your preferred programming text editor, and the `learncli` container.

2.7 Create directories with the `mkdir` program

When you work on a project and want to organize its files separate from others, you'll need to create a directory. Yes, *there's an "app" for that.* The CLI program to make a directory is `mkdir`. In the next few commands you will change your working directory to `/mnt/learncli/workdir`. Then, you will make a new directory named `ch2` for upcoming examples in this chapter. Finally, you'll change your working directory to be the directory you created. After making the directory, try looking at your host PC's `workdir` directory to see that a change made from inside your container is persisted outside of it via the shared directory.

```
learncli$ cd /mnt/learncli/workdir
learncli$ mkdir ch2
learncli$ cd ch2
```

In the commands you ran above, can you discern which parts of the commands were paths? Of those paths, which were *relative* and which were *absolute*?[3] After the second `cd` command, what is the absolute path of your current working directory? What program can you run to verify you are correct?

[3] The `/mnt/learncli/workdir` path was absolute while `ch2` was relative. The giveaway is the leading forward slash on the former and the lack thereof on the latter. The relative path `ch2`, in this example, is equivalent to the absolute path `/mnt/learncli/workdir/ch2`.

Open the manual page for `mkdir`. How can you tell the `DIRECTORY` argument is required while `OPTIONS` are not?[4] By default, the directory argument expects a path where the *dirname* directory already exists and the *basename* is the name of the folder being created. If you try creating a directory whose parent directory does not exist, you will receive an error message in your terminal.

2.8 Copy files with the cp program

Making a copy of a file on your file system is something you'll commonly do. The `cp` program is the universal "copy file(s)" program in command-line interfaces. Go ahead and give the `cp` program's manual page's *synopsis* and first line of *description* a quick read.

Now, let's try copying the dictionary file of words to your current working directory. Assuming you're continuing from the previous section, your current working directory will be `/mnt/learncli/workdir/ch2`. How can you confirm it is? If it is not, perhaps because you're returning to the text after previously `exit`ing your container, how can you change your container's working directory to the expected path?

```
learncli$ cp /usr/share/dict/american-english words
learncli$ ls
words
learncli$ cat words | less
```

Recall from the `cp` manual page that a common usage of `cp` takes the required arguments `SOURCE` and `DEST`. Both `SOURCE` and `DEST` are paths and in the example above, `SOURCE` is the *absolute* path `/usr/share/dict/american-english` and `DEST` is the *relative* path `words`. The result of running the command is a copy of the `SOURCE` file was made at the *absolute* path `/mnt/learncli/workdir/ch2/words`. Since it's within the shared directory, this file is now accessible from your host PC, as well. Mixing and matching absolute paths with relative paths, as shown above, is a common practice. Often you'll want to copy some file from outside of your project's working directory into it or out of it. Entire directories and their contents can be copied with `cp`, as well, using the `--recursive` option.

2.9 Silent success, noisy errors, and --verbose mode

Command-line programs whose purpose is to take an action saved to the file system conventionally display *no output* when successful. For example, both `cp` and `mkdir` offered you no output after running them despite working as expected. The motivation for this default behavior is to cut down on unnecessary noise. If you instruct a directory to be created or a copy to be made and you see no output, you can safely assume it worked.

You *will* receive output when these programs do not operate as you should expect them to, due to an error. For example, try copying the `words` file to a path whose *dirname* is nonexistent.

[4] Recall that a common pattern in manual pages is to surround optional arguments in square brackets.

```
learncli$ cp words /foo/words
cp: cannot create regular file '/foo/words': No such file or directory
```

When you need to know exactly what these "silently successful" programs are doing you can run them in a "verbose" mode. The `--verbose` argument is common across standard programs such as `cp` and `mkdir`. It tells the program, `cp` in this case, to print out the steps it is taking. Try following along in your container:

```
learncli$ mkdir --verbose a-sub-dir
mkdir: created directory 'a-sub-dir'
learncli$ cp --verbose words a-sub-dir/words
'words' -> 'a-sub-dir/words'
```

If you run a command that is silently successful yet it does not appear to have the effect you expected, try running it in a verbose mode for some insight on the actual actions it took.

2.10 Hidden "dot files" begin with a .

File names and directories which begin with a period, or a "dot", are conventionally considered *hidden* files. These files and directories are typically used to store the settings, preferences, and metadata of tools and projects. Try making a copy, with `cp` in verbose mode, of the `words` file to a file named `.words-copy`. Remember, do not forget the leading period in the file name indicating it is a hidden file.

```
learncli$ cp --verbose words .words-copy
'words' -> '.words-copy'
learncli$ ls
a-sub-dir words
```

When running the `ls` program, it would appear no copy was made, even though the output of `cp` running in `--verbose` mode confirmed *it was*. By default, hidden entries are not displayed when listing a directory's content using `ls`.

2.11 Long vs short options and case sensitivity

To learn how to ask `ls` not to ignore hidden entries, refer to its manual with the command `man ls`. In the description section, look just below the line "Mandatory arguments to long options are mandatory for short options too." The first two arguments you see listed are:

```
-a, --all
       do not ignore entries starting with .

-A, --almost-all
              do not list implied . and ..
```

The description of the first mode sounds *exactly* like how you want `ls` to run. Notice there are two variations listed above it. The `--all` variation is called a *long option* because it uses two dashes and a complete word follows. The `-a` is a *short option* because it uses a single dash and a single letter. Long and short options are *mostly* interchangeable. This text chooses long options because they read less cryptically than short ones. Experienced users and many online tutorials will choose short options because once you learn them they are faster to type.

The description of the second mode will make more sense shortly, but focus on its long and short options. When long options are made of multiple words they tend to be separated by dashes, such as `--almost-all`. The short option is `-A` is distinctly different from `-a` because it is capitalized. ***Program names*, *arguments*, and *paths* are all *case sensitive* when you work at the command-line.**

2.12 List hidden files with `ls --all` or `ls -a`

Try using the `--all` long option which tells you it will "not ignore entries starting with '.' ." Then try using the `-a` short option, too.

```
learncli$ ls --all
.  ..  .words-copy  a-sub-dir  words
learncli$ ls -a
.  ..  .words-copy  a-sub-dir  words
```

A ha! There's the hidden file `.words-copy`, but you'll also notice those strange `.` and `..` entries, too. We'll come back to those in the next section. Other than not showing up in `ls`, there's nothing special about hidden files. You work with them just the same as ordinary files. Try printing the contents of `.words-copy` and piping it into `less` for pagination[5].

2.13 Parent directory `..` and current directory `.` links

Two curious entries, `.` and `..`, were shown when you listed this chapter's working directory's contents using `ls` with the `--all` flag. What *are* these entries? Since each begins with a `.`, you know why both were hidden in previous `ls` commands. If you run `ls --all` in any other directory, you'll see every other directory has them, too.

These two entries are special and created automatically for you in every directory. They are both *links* to directories. We have not discussed a *link* yet, which is a third kind of file system entry besides a file or a directory. A link "points" to something else in the file system[6].

[5] Remember the `cat` program reads a file and prints it out, so you should have tried `cat .words-copy | less`

[6] If you have worked in a programming language with pointers, such as C, C++, or Rust, a file system link is much like a pointer. If you have experience in a memory managed language, such as Java, JavaScript, or C#, you can think of a link as similar to a reference.

The `..` link points to the directory's parent directory. You will frequently use this link in conjunction with `cd` to change your working directory to the current working directory's parent. Try the following example:

```
learncli$ cd a-sub-dir
learncli$ pwd
/mnt/learncli/workdir/ch2/a-sub-dir
learncli$ ls --all
.  ..  words
learncli$ cd ..
learncli$ pwd
/mnt/learncli/workdir/ch2
```

Since every directory automatically has both `.` and `..` entries, you can combine parent directory links into a relative path to move "up" in the file system hierarchy by more than one directory at a time.

```
learncli$ pwd
/mnt/learncli/workdir/ch2
learncli$ cd ../..
learncli$ pwd
/mnt/learncli
learncli$ cd workdir/ch2
learncli$ pwd
/mnt/learncli/workdir/ch2
```

A useful program for "expanding" a relative path to the absolute path it refers to is the `realpath` program. It takes no action besides printing out a *canonicalized* absolute path. Try it:

```
learncli$ pwd
/mnt/learncli/workdir/ch2
learncli$ realpath ..
/mnt/learncli/workdir
learncli$ realpath ../..
/mnt/learncli
learncli$ realpath ../../..
/mnt
```

The single dot `.` entry in a directory links to itself. It is useful when you want to specify the current directory as an argument to a program that expects some directory's path. For example, the `cp` program will copy a file from one directory to another and retain the same filename if you provide a directory as the `DEST` argument. You've already copied the words file into the current working directory using an absolute path, now let's try a more idiomatic way applying your knowledge of the single `.` link.

```
learncli$ cp /usr/share/dict/american-english .
learncli$ ls
a-sub-dir  american-english  words
```

Notice the second argument to `cp` was `.`, the link to the current directory. You can think of this command as, "copy the file at `/usr/share/dict/american-english` to the current working directory."

You can convince yourself the single dot `.` is a self-referencing link in every directory using `realpath`.

```
learncli$ pwd
/mnt/learncli/workdir/ch2
learncli$ realpath .
/mnt/learncli/workdir/ch2
```

You may encounter paths beginning with a `./`, such as `./words`. These are relative paths. The relative path `./words` is equivalent to the relative path `words`. Use `realpath` to convince yourself the leading `./` is redundant in a relative path.

```
learncli$ realpath ./words
/mnt/learncli/workdir/ch2/words
learncli$ realpath words
/mnt/learncli/workdir/ch2/words
```

2.14 Move or "rename" files with the `mv` program

With the `mv` program you can *move*, or "rename", a file or directory to some other *path*. Open the manual page for `mv` and read its synopsis and textual description. Just like `cp`, the program to copy files, `mv` has arguments for a `SOURCE` and `DEST`. Unlike `cp`, after the `mv` program successfully completes `SOURCE` will no longer exist as it has been moved to the path `DEST`.

Try moving the hidden file `.words-copy` to the non-hidden name `words-copy`.

```
learncli$ mv .words-copy words-copy
learncli$ ls --all
.  ..  a-sub-dir  american-english  words  words-copy
```

You can use `mv` to move a file from one directory to another. Try moving `words-copy` into the directory `a-sub-dir`.

```
learncli$ mv words-copy a-sub-dir
learncli$ ls
a-sub-dir  american-english  words
learncli$ ls a-sub-dir
words  words-copy
```

The example above moved the file `words-copy` into the `a-sub-dir` directory. If you specify a path to a directory as the second argument, then `mv` moves the original entry to the directory and retains its filename.

Check for Understanding: Move the `words-copy` file back into the current working directory[7]. (Hint: Consider using the `.` link.)

The single `mv` program is used to complete two distinct tasks you've done separately in a graphical file system explorer. First, where you may have "renamed" a file from one filename to another. Second, where you cut and pasted or dragged a file from one directory to some other directory, `mv` does this, as well. This unification is thanks to operating on *paths*.

2.15 List directories recursively with the `find` program

As a project grows in size, making use of many subdirectories to stay organized, it is tedious to use `ls` to list each directory's contents in search of some file. The `find` program is like `ls` in that it lists a directory's contents, but it also lists the contents of subdirectories recursively. If you open the manual for `find`, then you will see an optional argument named `starting-point`. The starting point is the directory you want it to traverse. Try `find` in the working directory `/mnt/learncli/workdir/ch2`.

```
learncli$ find .
.
./american-english
./words
./a-sub-dir
./a-sub-dir/words
./words-copy
```

Notice the filename `./a-sub-dir/words`, a copy of the `words` file made in the subdirectory `a-sub-dir`, is listed. Currently, there are not many files in our project, so try `find` with a starting point of the *root directory*. There are *a lot* of files in the `learncli` container to support the operating system and installed programs, so you should pipe the output to `less`. You do not need to scroll through them all and you certainly should not worry over their purposes.

```
learncli$ find / | less
... every file in the system ...
```

In the previous chapter you learned a superpower of the command-line is the ability to connect simple programs together. The `find` program produces output and the `grep` program *filters* output. As a challenge, can you find all files in your container's file system whose filename ends with "words"[8]?

2.16 Delete files with the `rm` program

When you no longer have a need for a file and want to delete it, use the `rm` program, short for "remove". Like `ls`, its manual provides a synopsis with an arbitrary number of `OPTION`

[7] `mv a-sub-dir/words-copy .`

[8] `find / | grep words$`

arguments and FILE arguments.

```
SYNOPSIS
       rm [OPTION]... [FILE]...
```

Try removing the file a-sub-dir/words relative to the current working directory of /mnt/learncli/workdir/ch2.

```
learncli$ ls a-sub-dir
words
learncli$ rm a-sub-dir/words
learncli$ ls a-sub-dir
```

As expected, the words file was deleted from a-sub-dir. The ls commands before and after were only to confirm the effect.

The rm program, by default, will not delete a directory. Confirm this by trying to delete a-sub-dir. The rm program *is capable* of deleting directories, but I'll leave you to reading its manual to learn how.

2.17 Delete directories with rmdir or recursive rm

2.17.1 Delete empty directories with the rmdir program

The program rmdir is for deleting, or "removing", empty directories. The word *empty* is important. If you attempt to delete a directory with files still in it, it will produce an error. Since the subdirectory a-sub-dir should be empty after the previous section's example, try deleting it:

```
learncli$ ls
a-sub-dir  american-english  words  words-copy
learncli$ rmdir a-sub-dir
learncli$ ls
american-english  words  words-copy
```

2.17.2 Delete non-empty directories with rm -i --recursive

When you are confident you want to delete a directory, which may have other files and directories as a part of it, the rm program has a --recursive mode which will traverse all subdirectories to delete all files. Until you are more comfortable, you should also use the -i, short for --interactive, mode so that you're asked to confirm each file deleted and have a chance to change your mind. Let's try deleting the ch2 directory entirely. Notice in the following example I am deliberately choosing to respond with n to the prompts about removing files and directories.

```
learncli$ cd /mnt/learncli/workdir
learncli$ rm -i --recursive ch2
rm: descend into directory 'ch2'? y
rm: remove regular file 'ch2/american-english'? n
```

```
rm: remove regular file 'ch2/words'? n
rm: remove regular file 'ch2/words-copy'? n
rm: remove directory 'ch2'? n
```

Be warned if you *do not* specify the `-i` flag you *will not be prompted* and the files will be deleted recursively. Try this out on the `ch2` directory to get us back to square 0 for `ch2`.

```
learncli$ cd /mnt/learncli/workdir
learncli$ rm --recursive ch2
learncli$ ls
```

I cannot emphasize how careful you should be when running the `rm` program recursively. Many people before you, myself included, have lost work to `rm`'s recursive mode. Nonetheless it is a necessary feature to use at times and is relatively safe if you run it interactively, the mode which prompts you to confirm removal of each file, using the `-i` argument.

2.18 Delete files and directories with *caution*

Files and directories deleted via the command-line are not recoverable in the ways you're accustom to, so delete carefully. When you delete a file on macOS it goes into your Trash, or on Windows your Recycling Bin, for some period of time before it is permanently deleted. When you delete a file from the command line it is gone.

Of course, you now have the knowledge to make your own "Trash" directory with `mkdir`. Rather than delete files or directories, you can instead move them to "Trash" with `mv`. Finally, when you know you no longer need any of the files, *then* you could permanently delete them with `rm`. This workflow emulates the Trash or Recycling Bin of your host PC. From a command-line interface you have the power to design your own workflows by piecing together simple tools.

2.19 Command Reference

Command	Description	Standalone Usage
`basename`	Print the target of a path	`basename [PATH]`
`cp`	Copy file(s) from SOURCE to DEST	`cp SOURCE DEST`
`dirname`	Print the path to parent directory	`dirname [PATH]`
`find`	Print the directory/file hierarchy	`find PATH`
`mkdir`	Create a directory	`mkdir [PATH]`
`mv`	Move or “rename” a file	`mv SOURCE DEST`
`realpath`	Expand a relative path to absolute	`realpath PATH`
`rm`	Delete a file or link	`rm PATH`
`rmdir`	Delete an empty directory	`rmdir PATH`