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