Command Line Interface Basics Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Given a folder structure diagram, a list of 'cd (path)' commands and target files, match the paths to the target files.
2. Create, rename, and move folders using unix command line tools.
3. Use grep and | to count matches of a pattern in a sample text file and save result to another file.
4. Find what -c, -r, and -b flags do in grep by reading the manual.
5. Identify the difference in two different files using diff.
6. Open and close nano with and without saving a file.
7. Use ‘curl’ to download a file.
8. Read the variables of $PATH.
9. Explain the difference between .bash_profile and .bashrc.
10. Create a new alias by editing the .bash_profile.
11. Given a list of common scenarios, identify when it is appropriate and safe to use sudo, and when it is a dangerous mistake.
12. Write a shell script that greets a user by their $USER name using echo.
13. Use chmod to make a shell script executable.

Navigating Your Filesystem

Imagine you're on a trip to a new country. You're carrying a dictionary, but it's slow to translate every word you hear and you need to use a map. You would get used to these limitations eventually, but wouldn't it be great if you spoke a bit of the local language instead?

Believe it or not, this is likely how you've been using computers for most of your life! Modern machines are built to make navigating easy and entertaining, but you're not "speaking the computer's language". That changes today. Let's explore the terminal!

We'll discuss:

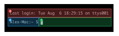
- How to find your way around your files *without*double-clicking any folders,
- Creating & managing new files & directories,
- and cleaning up after yourself.

Getting the lay of the land

You've already used a *terminal*for some tasks like controlling Git, but let's dive a little deeper. Your terminal is the interface you use to direct the computer. The word "terminal", as used here, comes from the early days of modern computing, when a *terminal interface*(often a screen & keyboard) would be hooked up to an otherwise manually-operated computer. This interface allowed a human to provide instructions to the computer without turning dials or requiring a complex manual to do so. Today, even though our terminal is built into our computer, we still use the term to refer to the application we're using to input our own instructions!

To keep everything in one place, we'll use the terminal that's built into Visual Studio Code. If you haven't yet, go ahead and open it. You can do so by clicking "View" from the top menu in VS Code, then "Terminal". You should see a new pane at the bottom of your editor.

Parts of the terminal



```
Last login: Tue Aug  6 18:29:15 on ttys001
Alex-Mac:~ $
```

1) The row in your terminal with a flashing *cursor* is called the *command line*. It's exactly as the name describes: the line upon which we enter our commands. Because we're using the command line to instruct the computer, we sometimes refer to the terminal as a *command line interface*or *CLI*.

2) The bit of text just before your command line is called the *prompt*. This will differ for each computer and will usually give you a little context about which directory you're in. You can customize your prompt to suit your style with custom code.

3) We refer to anything that's already been executed in the terminal as*output*. You'll likely see a little output in your terminal even if you haven't run anything yet. This is due to setup that's performed each time you begin a new terminal *session*.

A few quick tricks

Here are a few keyboard shortcuts to help you along. Some of these may be review for you. Give them a quick try before moving on.

- `Return/Enter`will submit the command you've typed.
- `↑/↓`will move up & down in your *command history*.
- Pressing `Ctrl + A`will move your cursor to the beginning of the line, while`Ctrl + E`will move you to the end.

- To clear the terminal screen, press `Ctrl + L`. You can still scroll up to see your previous output if you need to.

Understanding directories

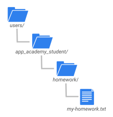
Let's run through a quick review of how your file system is structured. Your computer contains both *files*and *directories*. We distinguish these by their content: a file contains text or binary content that we can interact with, and a directory contains both files and other directories!

As an aside: It's easy to confuse "files" and "folders", so it's best to use the term "directory" instead. It's worth an extra syllable to prevent confusion!

Directories and files form a tree-like structure, where each directory creates a new branch and each file is like a leaf. We can write the *path*to the file or directory we want by joining all its *ancestors*with forward slashes, like so:

```
/users/app_academy_student/homework/my-homework.txt
```

Here's what that looks like in "tree" form:



There are a few special short names for particular directories you should know, too.

- `~`is your *home directory*. This is the same as `/Users/your_username/`on macOS.
- `/`is the *root directory*. This is the highest available directory and contains all the other directories & files in your file system.
- `.`is your current directory and `..`is your current directory's parent. For example, in `/Users/You/`, `.`refers to the `You/`directory and `..`refers to the `Users/`directory

Getting around

Now, we're comfortable on our command line. Let's start navigating our filesystem directly from our keyboard - no touchpad required!

Where am I?

When getting started in a new place, it's often helpful to orient yourself to your surroundings. The easiest way to orient yourself in the terminal is with the `pwd`command. `pwd`stands for "Print Working Directory". It will print the full path to your current directory out to your terminal. Give it a try now!

You might get back something unexpected here. If your prompt includes `~`, `pwd`will return `/Users/your-user-name/`in its place. Remember that `pwd`always returns the full path where you are, without any special characters or shortcuts.

Once you know where you are, it's good to see what else is there! We can look at what's present in the current folder with `ls`. `ls` is short for "List", and will display the contents of whatever path you provide it with. For example, you could run `ls .` to see your current directory's contents, or `ls ~/Projects`to see the contents of the "Projects" directory inside your home directory. When you don't provide any path, `ls`defaults to the contents of your current working directory.

A closer look at our contents

By itself, `ls` is useful but can be a little misleading. Linux & MacOS both support the concept of hidden files. These are files or directories whose names are preceded by a `.`. We've seen this before with the `.git`directory. Within a Git repository, `ls`alone won't display the `.git`folder at all. We'll see many more hidden files in upcoming lessons.

Command line instructions allow you to use *options*to alter their behavior. We set these options with either a single `-` (for shorthand options of one letter) or a `--`(for option *keywords*: whole words or phrases). Here's an example using short options with `ls`:

```
> ls -a -l
```

The above command runs `ls`, showing **a**ll files and displaying them in a**l**ist format. This ensures that we see **all**files, including those that are hidden. Viewing contents in a list format can make it a little easier to read, and it will show us some extra info about each file/directory! We'll dig into what that extra info means in a future lesson.

Here's one neat trick you'll see often with command line options: you can combine short options! Instead of typing `-a` and `-l`separately, you can run the same command this way:

```
> ls -al
```

Short options like this aren't order dependent, so `ls -la` will perform the same action.

Navigating directories

Now we know where we are and we know how to see what's around us. Let's set off on an adventure! It's time to navigate to other directories.

We switch to a different directory with the `cd`command. `cd`, which stands for "Change Directory", expects a path just like `ls`. Running `cd` with no arguments will assume you'd like to change to your home directory, and you'll end up back at `~`.

You can `cd`from any folder you have permission to access to any other folder on your system. There's no need to move in small steps! You can jump directly from `~`to `~/Projects/Homework-Week-1/Project-Name/code`with a single `cd`command.

Here are some short examples of common `cd`commands you'll use:

- Change to your home directory:

  ```
  > cd
  # OR
  > cd ~
  # OR
  > cd /Users/YourUserName
  ```

- Navigate up a single level from your current position:

  ```
  > cd ..
  ```

- Move back to the last directory you were in:

  ```
  > cd -
  ```

A caveat

You're browsing through your file system when you hit a snag: you encounter a "Permission denied" error. Oh no!

Have no fear. This is perfectly normal. Your operating system has a strict permissions system that tries its best to keep you from doing accidental damage. This is less obvious when you're browsing folders with Finder or File Explorer, where dangerous files/directories are hidden from view. In the terminal, though, these unexpected blocks can be jarring.

If you have a problem with "Permission denied", it's best to ignore it and go another direction for now. We'll discuss ways around this once you've had more practice in the directories you already have permission to access.

Making changes

You're a lean, mean, navigating machine! That's great, but now it's time to gain more control of your environment. Let's discuss how to create your own files and folders from the command line.

Creating new files & directories

You may not have much navigating to do if you're in an empty home directory. To start, let's discuss files. The fastest way to create an empty file is with the`touch`command. You give `touch`a path & file name, and it creates an empty file at that path with your given name. Here's an example:

```
> touch myApp.js

> touch ~/.js_settings
```

Note that `touch`doesn't put any content in the file, nor does it open the file for editing. This is a great utility for laying your files out, but you'll quickly want to move to a file editor (like VS Code) to make changes to these files.

For directories, we have the `mkdir`command. `mkdir`is short for "Make Directory" and will create a new, empty directory with the name you pass it:

```
> mkdir my-cool-projects

> mkdir ~/new-code
```

A common problem when learning `mkdir`is trying to create *nested*directories. For example, if I wanted to create a "first-week" directory inside a "homework" directory in my home folder, I would need to ensure the "homework" directory exists first. Here's what that looks like:

```
> mkdir ~/homework/first-week

mkdir: ~/homework: No such file or directory
```

We can solve this with a commonly-used short option for `mkdir`: `-p`. The `-p`option stands for **parent**, and it will cause `mkdir`to create *all*parent directories it needs to create the requested directory:

```
> mkdir -p ~/none/of/these/directories/exist/but/now/they/will
```

One last thing: when naming files and directories, do not use spaces! You can make multi-word names more distinct by using underscores, hyphens, and using camelCase. While files are *allowed*to have spaces in their names, this can complicate navigation. You'll thank yourself later if you avoid them altogether.

Manipulating existing files

You're browsing directories. You're making files. Woohoo! Are you ready to make some changes?

Just like using your mouse in Finder, you can copy/relocate files and directories from the terminal. The commands you'll need are `cp` and `mv`.

`cp` is short for "copy" and will create a duplicate of a file. It requires two arguments: a *source*and a *destination*. *source*can be the relative or absolute path of a file, *destination*can be a path to a file or a directory. If *destination*is a directory, `cp` will copy the source file into that directory. If *destination*is a file path, `cp` will copy the source file into that new location.

Gotcha: If a file already exists in the destination of your copy command,`cp` will overwrite the existing file.

```
# Will copy the file into the `people` subdirectory.
> cp best-friend.txt people/

# This command is identical to the above.
> cp best-friend.txt people/best-friend.txt
```

In each of these cases, we'll create an exact copy of `best-friend.txt`from your current working directory and place that copy in the `people`folder.

You can copy directories just like files, but you'll need a special short options to do so: `-r`. This option, short for "Recursive", copies not just the directory but all of its contents! Without it, the directory will fail to copy:

```
> ls
my_dir my_other_dir

> cp my_dir my_other_dir/
cp: -r not specified; omitting directory 'my_dir/'
```

Alternatively, `mv`"moves" a file from one place to another. Think of this like the "Cut" options on other operating systems. Again, you pass two arguments:

```
> mv breakfast-foods/cereal.txt anytime-foods/cereal.txt

# or identically:
> my breakfast-foods/cereal.txt anytime-foods/
```

What if you need to rename a file? There is no `rename`command in the terminal. Instead, you'll use `mv` to accomplish this:

```
mv speled-rong.txt spelled-wrong.txt
```

Like `cp`, `mv` can be used to move or rename directories. However, unlike `cp`,`mv`does *not*require a flag to do so. This is because the `mv`operation simply renames the directory, so we're not concerned about the contents within it.

Clean up: aisle `~`!

Okay, file system traveler. You've thrown `mkdir`and `touch`around for long enough! Let's discuss how we remove files and directories.

There are two removal commands in your terminal: `rm` and `rmdir`. The former is for files or directories, while the latter is for directories only. The use cases can be a little confusing, so let's look at some examples.

First, `rmdir`. This command, short for "Remove Directory" is only for removing an empty directory. If the directory has any files or other directories within it, the command will fail. You'll use this command occasionally for cleaning up extra directories you've created, but you're more likely to use the other removal command we mentioned.

```
# Remember, ~/my-app must be empty for this succeed!
rmdir ~/my-app
```

Last, `rm`. This command is short for "Remove" and can be one of the most dangerous tools in your arsenal. We've mentioned this before, but it bears repeating: *never use `rm` unless you're absolutely sure of what you're removing*! The terminal is often much less forgiving than the Finder app or Recycle Bin.

To use `rm`, you provide a filename or path. If you need to remove a directory along with all of its contents, you can use the `-r` short option, which will "Recursively" remove all files within the directory before removing the directory itself. Your terminal will guide you along this process - all you have to do is type "y" for "Yes" or any other key for "no" as it asks you about each file within the directory you're deleting. Once a file has been `rm`'ed, it's unrecoverable, so be *careful* about what you use `rm` on!

```
> rm file-we-dont-need.txt

> rm -r directory-full-of-files-we-dont-need/
```

Try practicing these new tools by cleaning up the mess you've made while experimenting. You'll get lots more practice using these as the days progress.

What we've learned

Navigating in the terminal is a little different than we're used to, but it's much faster to type commands than to drag a mouse! You should now have a greater mastery of:

- Exploring your file system via the terminal,
- Creating, updating, and removing files and directories,
- Using command line options to enhance our tools with new functionality.

Common Tasks On The Command Line

Let's go for a deeper dive into the tools we'll use day in and day out. We'll cover:

- Searching for files by content,
- Performing multiple commands in sequence,
- Terminal text editors & file comparison,
- and how to perform common web tasks from the command line.

`grep`marks the spot

One of the most common tasks you'll have is searching for a particular piece of code in a project. This might be to help diagnose unexpected behavior, or it might just be because you can't remember exactly where you left off! Either way, having a tool to help you find your way to a specific point in your code is critical to your workflow.

`grep` is a command line utility that was originally created in 1974 and stands out as a well-tested, reliable tool that does one thing well: text search. The name comes from the command sequence `g/re/p`, meaning "**G**lobally search for a **R**egular **E**xpression and **P**rint" (We'll discuss *regular expressions*in a future lesson). You can use `grep` to find text in a particular file or across multiple files in a particular directory! It's like using "Cmd + F" from the CLI.

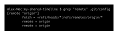
The simplest use of `grep` is with the contents of a single file. Here's an example of using it to find all the variables in a JavaScript file:

```
> grep var ./myAppFile.js
```

Notice that `grep`expects at least 2 arguments: a *pattern*and a *source* to search. In the above example, `var` is the pattern and `./myAppFile.js`is the source. If your search string is more than a single word, you'll need to wrap it in double-quotes:

```
> grep "I'm a programmer" ./resume.txt
```

By default, `grep`will return the whole line where your search string appears. This can be a little confusing at first:



```
Alex-Mac:my-shared-timeline $ grep "remote" .git/config
[remote "origin"]
        fetch = +refs/heads/*:refs/remotes/origin/*
        remote = origin
        remote = origin
```

In this example, there are four lines where the word "remote" appears in the file `.git/config`. That's not particularly intuitive from this raw output, though! Let's look at some ways to provide more helpful info from `grep`.

Common `grep`options

We'll occasionally need to modify `grep`'s default behavior. One common situation is when searching directories. It's likely that you'll want to use`grep` to find any files in a directory that contain a certain pattern. You can do this with the `-r`option, which stands for "**R**escursive". When run this way, `grep`expects a directory path as its source. It will search the directory and all of its children (files and subdirectories). Be aware: if there are lots of files and you're searching a common phrase, you might get back more than you expect with this option!

Another commonly-used option is `-n`, for "line **N**umber". This will show you the exact line for each match. Handy if you want to find something extra-fast!

By default, `grep` is case-sensitive, so searching for `Let`won't bring back instances of `let`. To override this behavior, use the `-i`option, for "**I**gnore

Case".

The last common option we'll discuss is `-c`, for "**C**ount". This will return only the number of matches, and not a full list of them. If you use this option in conjunction with `-r`, you'll see filenames for each count as well. This will be helpful when trying to track trends in code or when you need to know which directory contains the largest number of matches.

Teach yourself anything

`grep`, like most terminal commands, has many more options than we've discussed. How can we keep track of them all? Fortunately, we don't have to! Our system includes the `man`utility (short for **Man**ual) to help guide us when questions arise.

To learn more about any built-in command, just use `man`:

```
> man grep
```

This will open `grep`'s manual page, the official documentation for the command. You'll open in a *pager*, a lightweight document viewer meant to run in your terminal. To browse the `man`page, use your arrow keys to scroll up & down, and press the "Q" key on your keyboard to **q**uit.

`man`pages contain all the info you need to work with a command line utility. They're typically structured in a predictable way:

- A short summary & description of special features at the top,
- Command line options with explanations in the middle,
- And examples & cool facts (like when the command was created) at the bottom.

They are often terse & technical, but `man`pages are the official word on the tool you're curious about and are always a good place to start. Take some time to read through the `man`pages of some of the commands we've already covered!

Command redirection

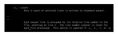
With both `man`and `grep`, we experienced some serious terminal overload. Wouldn't it be great if we could send that output somewhere else, like our text editor or a file to read later? We can do this via *command redirection*.

As the name implies, redirection is all about taking the output from one command and making it the input for a different command. Let's look at a very simple example using `|`, the *pipe operator*, where we'll combine `man`and `grep` to discover how to count pattern matches on a `man`page:

```
> man grep | grep -C1 count
```

Notice that we're not using the letters "L" or "I" but the vertical pipe character, found on the same key as `\` and above your "Return" key on US keyboard layouts. This operator takes the output from its left side and passes it to the command on its right as the final argument.

When we run the command above, we'll get back only the lines from `grep`'s `man`page that include the word "count", along with one line above & below for context:



```
     -c, --count
             Only a count of selected lines is written to standard output.

--
--
             Each output line is preceded by its relative line number in the
             file, starting at line 1.  The line number counter is reset for
             each file processed.  This option is ignored if -c, -L, -l, or -q
```

It's common to want to save output from a command into a file, too! We can do this a few different ways. The easiest is to use the `>`or `>>`operators:

```
# The single > operator will create a new file
# to place output in. Existing content will be overwritten.
> grep -r "TODO" my_app/ > my-app-todos.txt

# The double > operator will append your output
# to an existing file (or create a new one if needed).
> grep "my-name" list-of-names.html >> name-locations.txt
```

These redirection operators are lightning-quick, but have the caveat of not showing you the output before writing it to the given file! We can use the `tee`utility to both see our output and have it written to a file:

```
> ls my_directory/ | tee directory_contents.txt
```

Your command line supports its own scripting language and we're just scratching the surface with these redirection operators. Whenever you find yourself performing one or two simple tasks in the same order numerous times, consider how you might use redirection to simplify that process.

Editing files directly from the CLI

We can create files from the command line, search for content, and even combine commands into a single line! What about editing files directly from the command line too? Yup - there's a utility for that!

*In fact, there are a **large**number of text editors available for the command line. Two of the most popular ones you'll hear about are vimandemacs, which can both act as full development environments with a little customization! While you'll see lots of details about these editors online, it's best to avoid them for now. We'll focus on simpler editors with a much smaller learning curve.*

We're going to take a look at `nano`, a terminal text editor that provides easy-to-read instructions and is available on most systems. To get started, just type `nano`on your command line.

Let's take a look at the `nano`interface:

The upper part of `nano` is the editor body. Here you can type just like you would in any other editor. Text and linebreaks will work as expected, but support may be limited for fancy characters or keyboard shortcuts.

At the bottom of the screen you can see the command menu. These are the actions available to you and the keystrokes you need to access them. Remember that `^`represents "Ctrl" on your keyboard, so `^O` is the same as `Ctrl + O`.

When you attempt to save a file via `Ctrl + O`, you may be asked to confirm the filename. Type whatever name you'd like your file to have and hit "Enter" to save. Don't worry: if you've forgotten to save recently, `nano`will help you out by asking if you want to save unsaved changes before exiting!

When you run `nano`with no arguments, it opens to a new *buffer*, or empty space in memory ready for you to write. No files will be created until you save them. You can also run `nano`with a file path as an argument. `nano`will open the given file for you to edit:

```
> nano myApp.js
```

Why a terminal editor?

While `nano` is pretty stripped-down compared to a tool like VS Code, there are really great reasons to get familiar with it. The biggest benefits come when working in remote environments.

You'll sometimes need to log into a server somewhere else in the world and change a configuration file or run an update. If you need to edit text in that remote environment, it's unlikely you'll have access to VS Code. It's dramatically more likely that you'll have access to `nano`on your remote terminal. Now you can confidently edit files on computers you may never see in person! How cool is that?

Bringing the internet into your terminal

One last task you'll perform frequently on the command line is downloading files. This could be anything: an icon pack for your cool new app, an installation script for a larger program, or even a whole webpage to scrape for a project.

You can use the `curl`command to download from a URL to a file on your computer:

```
> curl https://www.my-website.com/my-file.html
```

With no other options, `curl`would download the contents of that URL to a new file titled `my-file.html`on our system. If you'd rather name the new file yourself, you can use the `-o`option:

```
> curl -o my-downloaded-file.html https://www.my-website.com/my-file.html
```

Like our other CLI utilities, `curl` is well-known and highly available. There's an extensive `man`page that's worth browsing through, too! `curl`offers lots of options that let you manage authentication, upload your own files, or even customize the type of request you're making. It's a powerful tool behind its rather simple facade.

What we've learned

Whew! If this feels like a lot of new tools, don't worry! They're all things you'll use on a daily basis. The beauty of using terminal tools like these is their simplicity: you can build complex workflows out of very simple utilities, and you get lots of practice in the process.

After reading this lesson and exploring on your own, you should be able to:

- find specific phrases in files and directories with `grep`,
- learn more about any command using `man`
- link commands together by redirecting output as input,
- edit file content using `nano`,
- and download files from the web using `curl`.

Understanding The Shell

As a developer, you'll spend much of your time working at the command line. Learning the commands you can use is important, but it's even more important to build a foundational understanding of where those commands go! Let's get under the hood and discuss the shell.

You'll learn:

- What a "shell" is,
- Shell-specific files and how to customize them,
- How commands are executed from the command line.

No turtles here!

We've used the word "shell" a few times with no context. Let's fix that now! A*shell* in computing terms is the software layer that translates your command line instructions into actual commands. Generally speaking, the shell serves two purposes: **Calling applications using your input**and **supporting user environments/customization**.

The shell is a small part of the much larger *operating system*, the software that sits between your input and the microchips inside your computer. The operating system, or *OS*, translates your actions all the way down into machine instructions that can be processed by your CPU. You've heard of the most popular OSes out there: Windows, macOS, and Linux.

Selling C shells by the seashore

Like most things in web development, many people have strong opinions about which shell is best. There are many shells available that you can install & use, each with its own idiosyncrasies.

In our lessons, we'll focus on two shells, the *Bash*shell and *Zsh*. Bash has been around for a little over 30 years and has been battle-tested on the most popular operating systems on the web. The biggest Linux operating systems all use it as the default shell, and macOS has used it as the default until switching to Zsh in macoS Catalina.

Zsh is now the default shell in macOS Catalina and has been in use since the 1990s and has a strong following amongst linux users.

You'll want to keep which shell you use in mind as you search for help with command line problems, as an answer intended for one shell may not work with your shell.

One nice thing about Zsh is it's scripting compatibility with Bash. Which means a shell script written for Bash will work in Zsh, although the opposite is not always true.

Shells vs Terminals

Your operating system may have an application called a "Terminal". It's good to note that a terminal is not a shell and a shell is not a terminal. Terminal applications are really emulating a piece of hardware known as a *terminal*which nobody really uses anymore.

So we emulate a terminal using an application on our computers. The Terminal application will then execute the shell and it will give us a prompt so we can then type commands.

Shell Prompts

You'll notice your shell prompts you to type something by putting either a `$` or a `%`before the cursor. Bash uses `$`by default while Zsh uses a `%`. But there's one more character you may see as your prompt character and that is `#`. This appears if you are logged in as `root`, the unix superuser. Since with great power comes great responsibility, whenever we see the `#`prompt we want to be careful what we type because we could delete system files or cause files to have the wrong permissions if we aren't absolutely sure what we are doing.

Two purposes

A shell's primary purpose is to interpret your commands. It does this by looking for applications installed on your computer and sharing any arguments or environment-specific data they need. Let's think about the following command, which will print the word "Test" to your terminal:

```
$ echo Test
```

In this command, `echo`refers to a program called "echo", while `Test` is an argument you've given the program. Bash doesn't know what `echo`does, only that it needs to find an application with that name and give it the word "Test". The shell trusts the application will handle this input appropriately.

Bash searches for applications using the `PATH`variable. `PATH` is a special variable that's available system-wide and includes all the directories you might store applications in. You can view your own `PATH`using that `echo`utility:

```
$ echo $PATH
```

You should see a list of directory paths separated by colons. When you run a command, your shell extracts the application name from the command and starts going through the `PATH`, directory by directory, looking for that name. Once it finds it, it passes along your input (or just starts the application, if there's no input available), and stops looking.

Note that we didn't just write `echo PATH`above. Instead, we included a `$`before the name of the variable. That's not a typo! On the command line, `$`before a word indicates that we're looking for a variable with the following name. Executing the command `echo PATH`would simply print out the word "PATH".

You might see a risk from this process right away: what happens if we have two different versions of the same program? Bash will call whichever version of the program it finds first in your `PATH`! If you're unsure which version of an application Bash is going to run, you can use the `which`command:

```
$ which echo
/bin/echo
```

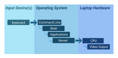
This output means Bash is going to run the application found at `/bin/echo`every time you enter a command beginning with "echo". If `which`returns a different version than you'd like to be running, you'll need to modify your`PATH`, which we'll discuss as part of our customization options.

Speaking of customizations: that's the shell's secondary purpose! Your shell makes things uniquely yours in a few different ways. The first is *environment variables*. These are variables that are stored in memory and made available to running applications by the shell. `PATH` is an example of an environment variable. Another is `HOME`, which stores the location of your home directory. You can see all your environment variables with the `env`command, but be prepared to scroll! The list can get pretty long.

Other customizations include scripts, command aliases, and your prompt. We'll discuss these more as we dig into deeper customization options in this and future lessons.

From the command line to the screen

It can be helpful to have a high-level overview of what's happening after you press "Enter" on the command line. Check out the diagram below to see how your command goes from keyboard to the monitor.



There are lots of extra steps we're overlooking here, but this should help you visualize the role of the shell.

Customizing your environment

Your shell's defaults (which may differ from system to system) are likely not doing a lot to help you. They're meant to get things up and going quickly, but you'll want to expand on them to suit your own tastes! Let's talk about how to make changes to your shell.

Startup files

The easiest way to make changes to the shell is directly from the command line. For instance, you can use the `export`command to change/initialize a new environment variable:

```
$ export NEW_VARIABLE=value
```

However, if you close your current terminal or open a new one, your environment variable will no longer be present! To persist environment variables and other customization settings, you'll need to put them in a file.

Bash supports several files intended for you to customize: `.profile`,`.bash_profile`and `.bashrc`.

Zsh supports several customization files as well, but we will only need to use`.zshrc`for most setups.

Each of these customization files are found in your home directory and are hidden (as indicated by the `.` at the beginning of filename).

These customization files are sometimes referred to as *dotfiles*.

These startup files are executed automatically at different times when you start your shell.

Bash startup files

The `.bash_profile`or `.profile`are executed when bash is started with a `-l` or `--login`command line flag. This is called a *login shell*. If you run bash without this commmand line flag it instead is a *non-login shell* and runs the `.bashrc`file.

To add a little complexity, Bash will run the `.profile`only if there isn't a`bash_profile`.

Now to confuse matters even more, often the `.bash_profile`will have a snippet of code in it that executes the `.bashrc`as well (this is the default on Ubuntu for instance)

To read more about which files Bash runs at which times at this link:https://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html

Zsh startup files

The `.zshrc`file is started anytime you start Zsh for both *login shells*and *non-login*shells. It does have a `.zlogin`file which is only executed on login shells but since `.zshrc`loads on both most Zsh users don't use `.zlogin`and instead put all their customizations in `.zshrc`.

To read more about all Zsh's various startup files you can follow this link:http://zsh.sourceforge.net/Intro/intro_3.html

To Login or Not to Login

So how can you tell if you have a login shell or a non-login shell? It depends on how your operating system works.

macOS

On macOS prior to Catalina if you open the Terminal application, it runs Bash as a *login shell* and therefore runs the`.bash_profile`on every terminal window you open.

After macOS Catalina, if you open the Terminal application it launches Zsh as a *login shell* and therefore runs both `.zlogin`and `.zshrc`.

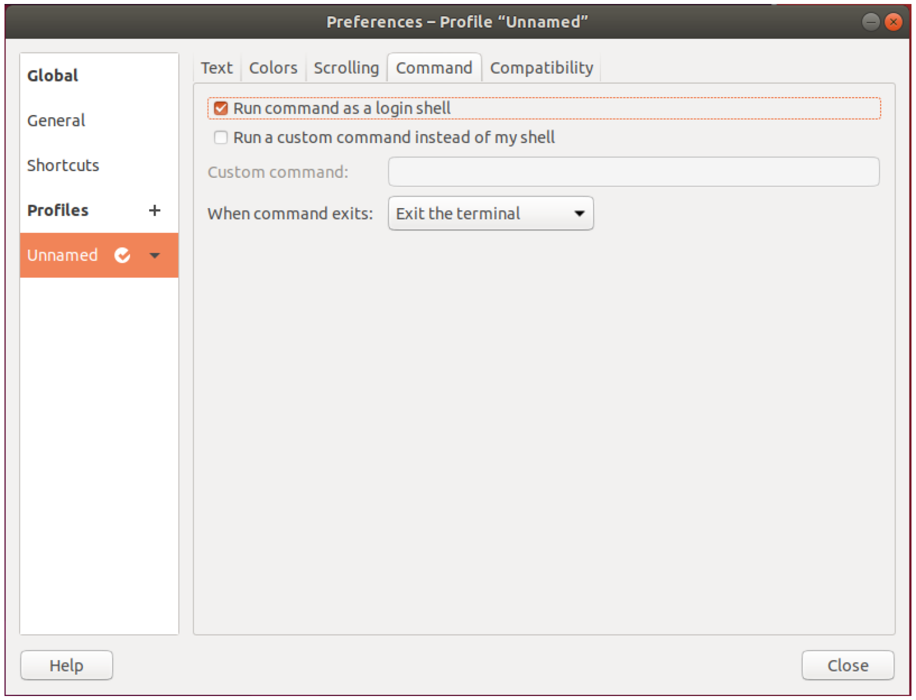
Windows using WSL

On Windows using WSL, the Windows Ubuntu terminal app runs every bash as a *login-shell* and therefore will run the `.profile`.

Ubuntu Linux

On Ubuntu Linux, by default the `.profile`is executed when you login to the Ubuntu Desktop. Ubuntu does not include `.bash_profile`by default. Then when you open the Terminal application, it runs each bash as a*non-login shell*which only runs `.bashrc`.

This can be somewhat frustrating to have to logout of the Ubuntu Desktop and login back in each time you change your startup file. So instead you can change the Terminal application to run each shell as a login shell by checking this checkbox in the Terminal preferences that reads *"Run command as a login shell"*


Preferences – Profile "Unnamed"

Profile: Unnamed · Tab: Command

- [✓] Run command as a login shell
- [ ] Run a custom command instead of my shell
- Custom command:
- When command exits: Exit the terminal

Visual Studio Code

When you use Visual Studio Code's integrated terminal, on macOS it defaults to being a login shell, while on Linux and Windows it does not.

You can change the integrated shell in vscode on Ubuntu and Windows using WSL by modifying or adding these settings in the VSCode `settings.json`file.

```
"terminal.integrated.shell.linux": "bash",
"terminal.integrated.shellArgs.linux": [ "-l" ]
```

If you use macOS Catalina, VSCode will still default to bash in it's integrated shell, so you can set it to use Zsh like so:

```
"terminal.integrated.shell.osx": "zsh",
"terminal.integrated.shellArgs.osx": [ "-l" ]
```

Whew this seems complicated, can you just tell me which file to use?

Sure, here's our recommendations on how to customize your system depending on platform to have the best results.

macOS Catalina with Zsh

- Put your customizations into your `.zshrc`file.
- Change Visual Studio Code to use `zsh`as it's shell.

macOS pre-Catalina with Bash

- Put your customizations into your `.bash_profile`file.

Ubuntu Linux

- Change your terminal to open login shells
- Put your customization into your `.profile`file.
- Change Visual Studio Code's integrated terminal to launch login shells as well.

Windows with WSL

- Put your customization into your `.profile`file.
- Change Visual Studio Code's integrated terminal to launch login shells as well.

Customization options

Whew, that's a lot of new jargon & theory! Let's look at some practical examples of what you can do with your dotfiles in Bash or Zsh.

Adding new options to your dotfiles starts with editing them.

You can open them directly from the command line using VS Code:

```
$ code ~/.bash_profile
# OR
$ code ~/.bashrc
# OR for Zsh
% code ~/.zshrc
```

Once you've saved your changes, you'll need to load any updated files into your shell. You can do ths with the `source`command. `source`will execute the file it's given, updating your currently-running environment with any changes you've made:

```
# For Bash
$ source ~/.bash_profile
```

OR

```
# For Zsh
% source ~/.zshrc
```

One of the most common customizations are environment variables. These should always be capitalized & use underscores instead of spaces to delineate words. Here are a couple examples of code customizing environment variables, along with comments explaining how each works.

```
# The simplest option: adding a totally new environment variable.
export FAVORITE_COLOR=blue

# Let's overwrite an existing environment variable with our own.
export HOME=/User/Student/Home

# Time to get get more creative: what if we want to _prepend_
# to the PATH variable, instead of overwriting it if it exists?
export PATH=/User/Student/Applications:$PATH
```

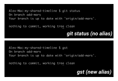
In that last example, notice how we used `$`before `PATH`. The dollar sign indicates that we're referencing a variable. Bash will replace `$PATH`with the value of the `PATH`variable, so we're effectively adding our own directory to the beginning of the `PATH`variable. You'll see this technique used a **lot** in dotfiles.

Another common customization is aliasing. An *alias*is a shorthand way of running a command. You might alias because a command is very long to type, or to modify the system's default behavior! Here are some examples:

```
# Here's a Git alias that will save you a few keystrokes.
alias gst="git status"

# Some more Git magic: show the short log with an even shorter command!
alias glog="git log --oneline"

# By default, 'rm' will remove the file you pass it. The '-i'
# option makes 'rm' ask you "Are you sure?" before removing the
# given file. This is a great safety net to have while you're learning!
alias rm="rm -i"
```

Don't forget to `source`your dotfiles if you're following along!

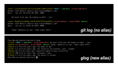
Entering an alias on the command line makes the aliased command act just like the full command was entered. You can still pass arguments like you normally would! Here are some "before & after" examples of our aliases in action:



```
Alex-Mac:my-shared-timeline $ git status
On branch add-mars
Your branch is up to date with 'origin/add-mars'.

nothing to commit, working tree clean
```

```
Alex-Mac:my-shared-timeline $ gst
On branch add-mars
Your branch is up to date with 'origin/add-mars'.

nothing to commit, working tree clean
```

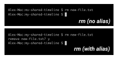
Notice how we get the same output from `gst` as `git status`with significantly less typing.



```
commit [illegible] (HEAD -> add-mars, origin/add-mars)
Author: Alex Martin [illegible]
Date:   [illegible]

    10 years from now: No humans on Mars - yet!

commit [illegible] (origin/master, origin/HEAD, master)
Author: Alex Martin [illegible]
Date:   [illegible]

    Today: Humanity on top - what comes next?
```

```
Alex-Mac:my-shared-timeline $ glog
[illegible] (HEAD -> add-mars, origin/add-mars) 10 years from now: No humans on Mars - yet!
[illegible] (origin/master, origin/HEAD, master) Today: Humanity on top - what comes next?
[illegible] 2.5 million years ago: The dawn of man.
[illegible] 4.5 million years ago: Our favorite zoo animals, just...bigger.
[illegible] 66 million years ago: Asteroid impact & mass extinction.
[illegible] 200 million years ago: Dinosaurs roam.
[illegible] 4.5 billion years ago: The Earth is formed.
Alex-Mac:my-shared-timeline $
```

Our `glog`alias provides us with a short, quick-to-read commit history that takes even fewer characters typed than the default `git log`!



```
Alex-Mac:my-shared-timeline $ rm new-file.txt
Alex-Mac:my-shared-timeline $
```

```
Alex-Mac:my-shared-timeline $ rm new-file.txt
remove new-file.txt? y
Alex-Mac:my-shared-timeline $
```

The `rm`alias is an example of changing the default behavior of a command. Before we aliased `rm`, removing a file happened with no confirmation at all. After the alias is applied, we see a prompt that lets us respond "y" (for "**y**es") or "n" (for "**n**o"). Think of this as the command line version of a "Confirm" or "Cancel" pop-up window!

What we've learned

We're diving deeper behind the scenes of our terminal & operating system, and hopefully you're beginning to understand how things work together! You should now be comfortable with:

- What a "shell" is in computer terms,
- explaining, at a very high level, how your computer interprets your commands,
- reading and redefining the `PATH`environment variable,
- and customizing your system via *dotfiles*

Bash Permissions & Scripting

Now that you're familiar with Bash, let's get back to what we're all here for: PROGRAMMING! We'll discuss the basics of Bash scripting, including:

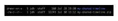
- Creating a new script file,
- Accepting user input and conditionally responding to it,
- Modifying permissions to make the script executable,
- and choosing how to execute your script (or any other application).

Understanding `sudo`and file permissions

Before we can write any code, we need to get a quick look at file permissions in UNIX-based operating systems. Without understanding this critical part of file management, you'll have a difficult time creating new scripts of your own.

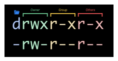
Remember the command/option combo `ls -al`? We'll focus in particular on the `-l`command line option, which **l**ists the files in the directory along with their metadata. This includes their *file permissions*(also sometimes referred to as *modes*). Permissions determine who can access a given file or directory.

Here's an example of what a directory and file look like via `ls -al`:



```
drwxr-xr-x   5 jdh  staff    160 Jul 28 19:38 my-shared-timeline
-rw-r--r--   1 jdh  staff  32243 Jul 30 13:12 my-shared-timeline.zip
```

The file permissions are the ten characters on the left side of each line. Let's break them down:



```
drwxr-xr-x
-rw-r--r--
```

The leftmost position is the *directory indicator*. This is the easiest part to read! You'll see a `d` for directories and a `-`for files.

The remaining nine characters are broken into groups of three, representing the*owner*, *group*, and *others*from left to right. Each group has three permissions available:

- *read (r)*: view a file or directory's contents
- *write (w)*: modify a file or directory's contents
- *execute (x)*: run a file like an application, navigate into a directory

A letter in any position means that permission is *granted*, while a `-`means the permission is *restricted*.

In our example above, we can see that the owner (`jdh`) of the file`my-shared-timeline.zip`can view the file or edit it, but the group (`staff`) and world (everyone else) can only view it. No one is allowed to execute the file, but that makes sense in this case: `x`permissions are mostly used by scripts/applications and directories.

Numeric permission notation

While the letters `r`, `w`, and `x`are easy to read, you'll also see a numeric notation for file permissions. To convert letter notation to numeric, you'll need to grant each permission a number value. `x` is worth 1, `w` is worth 2, and `r`is worth 4. Now, tally up points for each group and write them out side by side.

From our example above, we'd get `644`. This is the numeric notation for the permissions of `my-shared-timeline.zip`. You'll sometimes see these numeric formats referenced in documentation or Bash error messages, so it's good to know how to read them even if you don't use them often!

Numeric permissions are presented in *octal notation*. You can read a few more details about how this works on LinuxCommand.org.

Modifying permissions

We're not stuck with the permissions a file starts with! We can use the `chmod`command to update the permissions of a file ourselves. `chmod`, short for "**ch**ange **mod**e", dates back to the early days of UNIX. You may hear it pronounced as "cha-mod", "see-aych-mod", or even referred to as "change mod".

Let's say we've written a cool "How To" guide that we'd like to share with all users of our system. Assume we're starting with permissions where the owner can read or update the file but no one else can even read it (`-rw-------` or`600`).

To change this so that anyone on the system can read it, we could run:

```
> chmod +r my-guide.txt
```

This is saying "Add the 'r' permission for all users who try to access`my-guide.txt`.

To *revoke*that permission, you can exchange `+` for `-`:

```
> chmod -r my-guide.txt
```

Uh oh! If we check `my-guide`now, we'll see that we're not back at `-rw-------`

- we're now at `--w------`! When you change permissions using letter notation and don't specify a target group, the change affects **all three**groups at once. You can scope this down by preceding the operator with either `u`(for**user**, meaning the file's owner), `g`(for the file's owning **group**), and `o`(for **others**).

Let's add read permissions back for the owner:

```
> chmod u+r my-guide.txt
```

Perfect! Now we've reverted our permissions back to their original state.

Ignoring permissions entirely with sudo

Most systems include a *root* user that has total authority. The root user can run applications and change files indiscriminately. With that much power, it's hard to keep a system safe! For this reason, it's a bad practice to log in as`root`.

To keep us from having to memorize both our own password and that of the `root`account for every system, we've got a helpful command called `sudo`. `sudo` is short for "**S**uper **U**ser **Do**", and as the name implies it allows you to impersonate the `root`user for a particular task.

`sudo` **is inherently dangerous.**Every time you use it, you're at risk of doing real damage to the system you're on. While there are some advanced safeguards & security featuresfor the `sudo`command, it's best to use is sparingly if at all.

Generally, applications will provide a message letting you know if `sudo` is needed. This might include installing new applications or browsing system configuration files. These decisions are only as safe as you make them, so be sure you're confident about the changes you're making before using `sudo`.

There's a common trap we've mentioned before where `sudo`comes up frequently: the `rm`command. Using `rm`indiscriminately as an unprivileged user can cause you some frustration but is unlikely to do any permanent damage. Using `sudo rm` in any capacity can wreak havoc on a system and may result in significant data loss.

Never use `sudo`with `rm` or with code from the internet that you don't understand!

Bash scripting

Let's talk about scripting. We've used the word "script" quite a few times already, but what exactly is it? A *script* is simply a text file that we've granted permission to execute on our system.

You'll have lots of opportunities to run scripts for everything from setting up your environment on a new computer to installing new applications. Writing your own scripts is a great way to automate repetitive tasks.

Script requirements

An effective script requires three things:

- An interpreter directive
- A commented description
- A script body

The *interpreter directive*(more commonly called the *shebang*) is the first line of the script file. It's used by the operating system to know which application should be used to run your code.

Why "shebang"? This name is a combination of two words: hash, a reference to the octothorpe symbol (`#`) and bang, a reference to the exclamation point (`!`). Helpfully, the "shebang" nickname will also help you remember in which order these characters are expected!

Here's an example shebang for a Bash script:

```
#!/bin/bash
```

Adding that line to the top of your script file lets the system know that you'd like to use the application `/bin/bash`to run your script file. As you learn more scripting languages, you can change change the shebang to make sure the script's language matches its executing environment.

The commented description is a best practice. Your script won't fail to run without it, but saving scripts without comments is a dangerous game - if you forget what it is or how it works, you won't have any reference to help re-learn it! Investing a few minutes in a good comment explaining why you're writing this script will save you a few of hours of headache down the road. Comments in Bash scripts must be preceded by an octothorpe (`#`) on each line.

The script body is where the magic happens. Here, you'll write commands just as you would enter them on your command line, and they'll be sequentially executed to complete the script. Each line should include a separate command.

A sample script

Here's a very simple "Hello World!" application in Bash:

```
#!/bin/bash

# "Hello, World!" by Alex Martin
#
# Prints a friendly message to the screen

echo "Hello, World! "
```

Notice that we've got all three of our key ingredients for a successful script. All we need is to copy this script into an empty file and make it executable.

Updating & running a script

Once you've written your script into a new file, you need to make that file executable. For security, your system won't run just any file! You'll need to mark it "safe" through the magic of `chmod`.

Assuming our script is called `hello-world`, this will make the file executable:

```
> chmod +x hello-world
```

We can now run it simply by invoking the file:

```
> ./hello-world
Hello, World!
```

Notice how we used `./`before the script. This isn't strictly necessary, but it's a good habit to get into. If we ran just `hello-world`and happened to have an application called "hello-world" available in our `PATH`, we might never make it to our own script! Preceding the script name with `./`ensures we're going to the run the script with that name in the current directory.

What we've learned

We've covered some of the basics of files and security, and we've looked at Bash's native scripting support. After this lesson you should be able to:

- explain the basics of file permissions from `ls -l`output,
- read and modify file permissions from the command line,
- create and a run a small Bash script.