

Overview Of The Modules

🕒 10 minutes

✅ Completed

Again, welcome! We're really happy that you're here! You're going to spend about 40 - 50 hours of your time working through this material and finishing some projects. We'll grade those projects and, if you pass them all, then you'll be eligible to enroll in and attend the full-time immersive course! This is really exciting.

The goal of this prep work is to get you ready in two ways for your App Academy learning journey.

1. The skills that you learn in this are skills that all full-stack software developers use *every day* of their lives.
2. The habits that you establish in this are the bases of the independence and self-reliance you'll need to get through the boot camp and through your career as you become a professional software developer!

But, don't be afraid. There's a lot of information found in this course. There's even more information found on the Internet. In each module, you'll have some instructions about the best ways to get help from the Internet with links to good documentation, good message boards, and good Q & A sites.

Here's a description of the modules in this prep work. Please read each of the descriptions so that you know which ones that you should add to your

journey.

Getting Your Computer Ready

You may be using a MacBook Air. You may be using a ThinkPad X1. In either case, you will need to get your computer ready for the upcoming work! That means some installation of (possibly) new software. The videos will show you an example installation. The readings will provide you with step-by-step instructions that the videos followed so that you can do the installation without having to pause and start the video over and over. The resources section provides you with the links that you can use to find the products and documentation for each of the pieces of software that you'll install.

Get Your Mac Ready

If you're using a computer made by Apple running macOS, then you'll need to go through this section to get everything installed and ready on your computer. You will need to closely follow the command-line instructions because you will be using a utility named *homebrew*.

Get Your Windows Ready

If you're using a computer made by HP or Dell or some other manufacturer running Windows 10, then you'll need to go through this section to get everything installed and ready on your computer. You're going to visit different parts of Windows that only programmers and folks like the Geek Squad usually take advantage of. One of the real joys of being a programmer

is learning all of the different ways you can improve your computer and, also, break your computer!

Get to Know Your Computer

You've probably double-clicked on a folder to look at the files inside it, whether those are photos you took or spreadsheets or PDFs of stuff that you've downloaded. Now, you'll start interacting with your computer like a software programmer rather than like someone that uses it for only email and Word documents. You'll spend time using the *command line* to issue commands to your computer rather than clicking on icons to run applications.

Learning Modules

Each of the following modules are the ones that you'll have to pass to get into App Academy. That may sound a little scary. But, if you study hard, do all of the projects and quizzes, you should be able to pass the tests at the end of each module.

You can learn and take these modules in any order. They are independent of one another. Each module challenges your brain and understanding in a different way. HTML and CSS challenges you to think in different languages to articulate a visual design. The Git module dares you to understand how to manage the chaos of change. The Boolean algebra and digital logic section tests your ability to dive into symbolic mathematics. All of these are skills that you need every day for programming on the Web. All of these are skills that you need every day to become a better programmer.

HTML and CSS

These are the basic languages of the World Wide Web! If you browse a Web site, your computer is getting HTML and CSS from another computer to show the pretty (or not so pretty) Web pages out in the Universe. In this module, you will get to learn those languages to make pretty Web pages yourself!

Git

When you start working with your code in class, you're going to want to keep your code safe. When you start working with other people, you'll also want to be able to track the changes made by each team member. With multiple people changing the same code at the same time, you'll want to make sure that the changes get merged together properly. This is what Git allows you to do. It's a powerful tool that can seem overwhelming. This module shows you how to interact with Git from the command line to manage your code.

Boolean Algebra and Digital Logic

Computers are pretty dumb. They can do seemingly amazing things. But, they really only understand a very limited number of instructions. At the very bottom of it all, computers understand zero and one, true and false. Boolean algebra is a way to create expressions that describe combinations of true and false. Digital logic is the way that computers take those Boolean algebra statements and put them into practice in the electronic circuits! When you get done, you'll have a sense of how computers work at the microscopic level!

Where Do You Go Next?

Depending on what kind of computer you have, please choose the proper section of **Getting Ready To Install** for your **Mac** or your **Windows** computer.

Installing Xcode, Homebrew, VS Code, Chrome, And Git

🕒 30 minutes

Set-Up Software On Your Mac

Watch the videos that show you how to do a successful installation.

- [Getting Ready To Install](#)
- [Installing Homebrew and Xcode](#)
- [Installing VS Code and Google Chrome](#)
- [Installing And Configuring Git](#)

Now that you've seen it, you can believe that it can be done!

1. Activate Spotlight (Command + Space) and type "terminal"
2. Select "Terminal" to start Terminal
3. Open Safari and navigate to <https://brew.sh>
4. Copy the command to be pasted into the Terminal
5. Paste the command into Terminal and press Enter
6. Press Enter when prompted
7. Enter your password when prompted

8. Wait a long time
9. Type `brew doctor` and hit Enter. If it reads that "Your system is ready to brew.", then you are done. If not, address the errors that you see.
10. Type `brew cask install visual-studio-code` and press Enter
11. Type `brew cask install google-chrome` and press Enter
12. Click on Finder
13. Click on the Applications section
14. Confirm that Google Chrome is installed
15. Confirm that Visual Studio Code is installed
16. Focus on Terminal
17. Type `brew install git`
18. You need to configure Git, so type `git config --global user.name "Your Name"` with replacing "Your Name" with your real name.
19. You need to configure more Git, so type `git config --global user.email your@email.com` with replacing "your@email.com" with your real email.

Congratulations! You are now ready to start learning!

Installing Ubuntu, VS Code, Chrome, and Git

🕒 30 minutes

✅ Completed

Set-Up Software On Your Windows Computer

Watch the videos that show you how to do a successful installation.

- [Getting Ready To Install](#)
- [Install WSL and Ubuntu](#)
- [Installing VS Code](#)
- [Configuring Git](#)

Now that you've seen it, you can believe that it can be done!

1. In the application search box in the bottom bar, type "PowerShell" to find the application named "Windows PowerShell"
2. Right-click on "Windows PowerShell" and choose "Run as administrator" from the popup menu
3. In the blue PowerShell window, type the following:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
```

4. Restart your computer
5. In the application search box in the bottom bar, type "Store" to find the application named "Microsoft Store"
6. Click "Microsoft Store"
7. Click the "Search" button in the upper-right corner of the window
8. Type in "Ubuntu"
9. Click "Run Linux on Windows (Get the apps)"
10. Click the orange tile labeled "Ubuntu"
11. Click "Install"
12. After it downloads, click "Launch"
13. Wait for it to install the local files
14. When prompted to "Enter new UNIX username", type your first name with no spaces
15. When prompted, enter and retype a password for this UNIX user (it can be the same as your Windows password)
16. Confirm your installation by typing the command `whoami` followed by Enter at the prompt (it should print your first name)
17. You need to update your packages, so type `sudo apt update` (if prompted for your password, enter it)
18. You need to upgrade your packages, so type `sudo apt upgrade` (if prompted for your password, enter it)
19. You need to configure Git, so type `git config --global user.name "Your Name"` with replacing "Your Name" with your real name.
20. You need to configure Git, so type `git config --global user.email your@email.com` with replacing "your@email.com" with your real email.

21. Open Microsoft Edge, the blue "e" in the app bar, and type in <http://chrome.google.com>. Click the "Download Chrome" button. Click the "Accept and Install" button after reading the terms of service. Click "Save" in the "What do you want to do with ChromeSetup.exe" dialog at the bottom of the window. When you have the option to "Run" it, do so. Answer the questions as you'd like. Set it as the default browser.

Congratulations! You are now ready to start learning!

Configure Your Text Editor

🕒 5 minutes

✅ Completed

Configure Your Text Editor

It is *vital* important that you exactly follow these steps to make sure that your edited files meet the expectations of the exercises that you'll finish. Without doing these steps, there is a likely chance that you will waste failures on file content problems even though you're doing everything right.

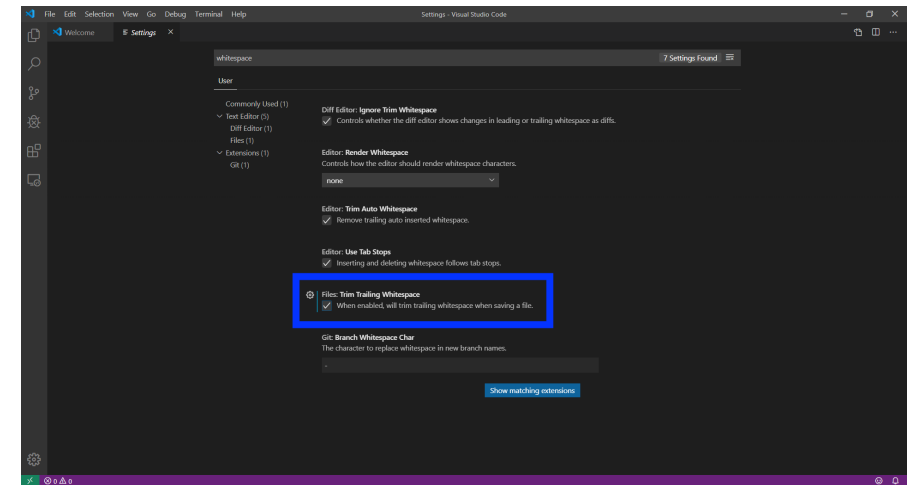
Open Visual Studio Code.

- On Windows, type "code" in the application search bar and click on the "Visual Studio Code App" result
- On macOS, hold down the Command key and press the space bar. Release the command key. Type "code". Choose "Visual Studio Code" from the Spotlight list

Once Visual Studio Code has opened, open preferences.

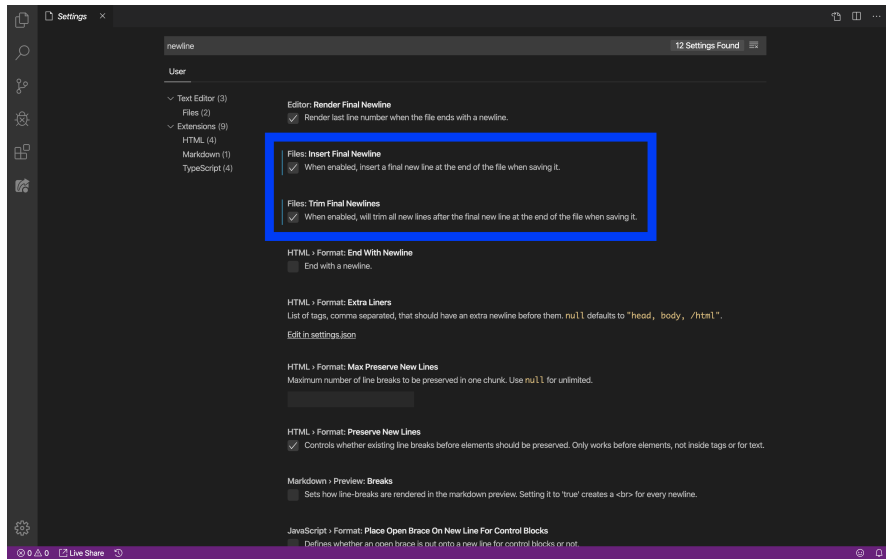
- On Windows, hold down the CTRL key and type comma. Release the CTRL key.
- On macOS, hold down the Command key and type comma. Release the Command key.

In the search bar of the *Setting* tab type "whitespace". Find the setting named "Files: Trim Trailing Whitespace" and enable it by checking the box.



Clear the search bar of the *Setting* tab and type "newline".

- Find the setting named "Files: Insert Final Newline" and enable it.
- Find the setting named "Files: Trim Final Newlines" and enable it.



Command Line Reference

✔ Completed

Command Line Reference

While getting to know the different ways to interact with your command-line environment, Ubuntu on Windows and Terminal on macOS, it's handy to have a reference lying about. This can be that reference!

You can also download this [one-page PDF](#) if you're the type of person that likes to have it available that way, too.

Summary of basic commands

The following table lists the commands to perform for different types of actions on files and directories in the Terminal/Ubuntu

Action	Files	Directories
Copy	cp	cp -r
Create	code	mkdir
Delete	rm	rmdir, rm -r
Inspect	ls	ls
Move	mv	mv
Navigate to		cd

Action	Files	Directories
View content	cat	ls

Glossary of terms

Here's a table of terms that you will want to remember not only as you work through this prep work but also during the remainder of your programming career.

Term	Definition
absolute path	A path that refers to a particular location in a file system. Absolute paths are usually written with respect to the file system's root directory, and begin with either "/" (on Unix) or "\ " (on Microsoft Windows). An example of an absolute path is <code>/home/appacademy/Downloads</code> which would be the path for the directory that contains downloads for the "appacademy" account.
command-line interface	A user interface based on typing commands that interacts with files and the operating system. On Windows, you will use the Ubuntu environment for that. On macOS, you will use Terminal.
current working directory	The directory that relative paths are calculated from; equivalently, the place where files referenced by name only are searched for. The current working directory is usually referred to using the shorthand notation "." (pronounced "dot").
file system	A set of files and directories. A file system may be spread across many physical devices, or many file systems may be stored on a single physical device; the operating system manages access.
filename extension	The portion of a file's name that comes after the final "." character. By convention this identifies the file's type: .txt means "text" file, .png means "Portable Network Graphics" file, and so on. These conventions are not enforced by most operating systems: it is perfectly possible (but confusing!) to name an MP3 sound file "homepage.html" which makes it look like an HTML file for showing content in a Web browser!

Term	Definition
home directory	The default directory associated with an account on a computer system. By convention for your account, all of your files are stored in or below your home directory. On macOS, that is found in the <code>/Users/«your-name»</code> directory. On Ubuntu, you can find your home directory under <code>/home/«your-name»</code> . On Windows, you will find it under <code>C:\Users\«your-name»</code> . On Ubuntu and macOS, the shortcut for your home directory is <code>~</code> .
operating system	Software that manages interactions between users, hardware, and software processes. Common examples are Linux, macOS, and Windows.
parent directory	The directory that "contains" the one in question. Every directory in a file system except the root directory has a parent. A directory's parent is usually referred to using the shorthand notation <code>..</code> (pronounced "dot dot").
path	A unique description that specifies the location of a file or directory within a file system. There are two kinds of paths: absolute paths and relative paths.
relative path	A path that specifies the location of a file or directory with respect to the current working directory. Any path that does not begin with a path separator character, <code>/</code> on Ubuntu and macOS, is a relative path. The path <code>Documents/Contracts/</code> is a relative path from the current working directory into a "Contracts" subdirectory under a "Documents" subdirectory. You know that "Contracts" is a subdirectory because there's another path separator (<code>/</code>) after it.
root directory	The top-most directory in a file system. On macOS and Ubuntu, its name is <code>/</code> .
shell	A command-line interface such as Bash or Z shell that allows you to interact with the operating system.
subdirectory	A directory contained within another directory. All directories except the root directory are subdirectories.
wildcard	A character used to specify a pattern to match. In Bash or Z shell, the wildcard character <code>*</code> matches zero or more characters. This means that <code>*.txt</code> would match any name that ends with <code>.txt</code> .

Regarding file extensions: since many applications use filename extensions to identify the MIME type of the file, misnaming files may cause those applications to fail. If you copy a picture from your phone to your computer and rename it "recipes.docx", then your word processor program will probably try to open it and complain that the file is unrecognizable or

corrupt. You know that it's just an image, though, because you're smarter than the computer.

Introducing the Shell

🕒 5 minutes

✅ Completed

Introducing the shell

Humans now interact with computers through a variety of communication methods: by typing on a keyboard, by pointing with a mouse, by touching a screen, and by speaking to it. The most widely used way to interact with personal computers is called a **graphical user interface**(GUI). With a GUI, we give instructions by touching portions of the screen or clicking a mouse to interact with elements of the GUI.

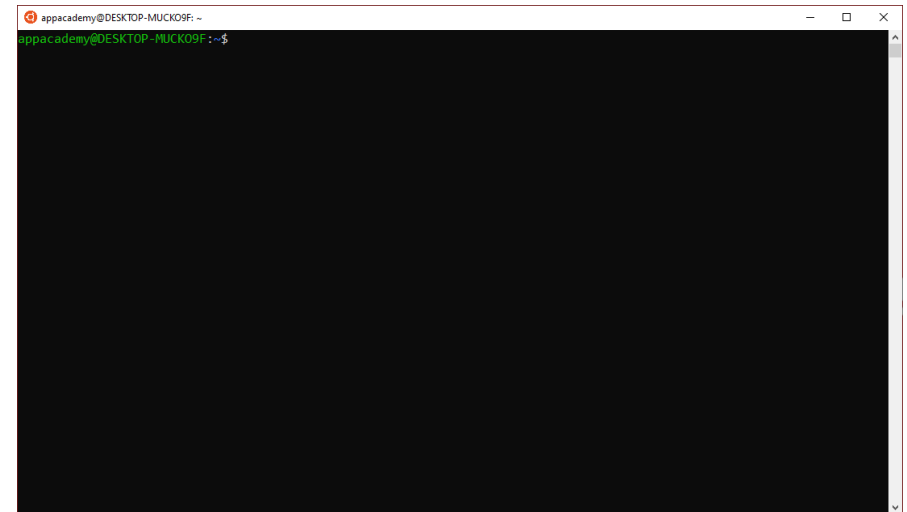
While the visual aid of a GUI makes it easy to interact with a computer, this method of interaction scales very poorly. Imagine that you had to search for some text in a thousand Word documents and paste the results into a single file. Using a GUI, you would point and click until your mouse wore out! Slogging through all of that, you would most likely miss some stuff, too.

This kind of task is super easy when you use a **shell** to interact with those files. The shell provides a way to *type* commands to the computer for it to execute. With the proper commands, you can use the shell to repeat tasks as many times as you want, including searching thousands of files.

As a developer, you will use the command line to launch programs, interact with files, and share code with other developers.

The shell

The shell is a program where you type commands. On Ubuntu for Windows, it looks like this.



With the shell, you can invoke complicated programs like crunching data to find a cure for cancer or simple commands that create an empty directory. On Ubuntu, the most popular Unix shell is named "Bash". On macOS since Catalina, Z shell is the default shell. Either way, it doesn't matter because the utilities that you would use to interact with the operating system remain the same thanks to common specifications created by people and companies that wanted interoperability.

Using the shell will take effort and time to learn. A graphical user interface presents you with choices to select. In a command-line interface, choices are not automatically presented to you. You start by learning a few commands.

What's really cool is that you can use the command line to interact with remote machines and supercomputers. Familiarity with the shell is near

essential to run a variety of specialized tools and resources including high-performance computing systems in the cloud. You can build on the command-line skills covered here to tackle a wide range of programming tasks. You can even change your identity so that you can get superpowers on the computer!

Getting started

When you first open your terminal, the shell presents you with a **prompt**. This indicates that the shell is waiting for your command. On Ubuntu for Windows, you would see something like

```
appacademy@DESKTOP-MUCK09F:~$
```

On macOS with Z shell, you would see something like

```
appacademy@demo ~ %
```

These are the prompts. The prompt contains some information that you may want to know about the session that you're using. Here are the different parts of each of those prompts:

- **appacademy** is the name of the user that's using the terminal. Normally, that will be your user name unless you decide to impersonate another user to do tasks that only they can do.
- **DESKTOP-MUCK09F** and **demo** are the names of the computers.
- **~** is the alias, the nickname, for you home directory.
- **\$** and **%** are the indicators used by Bash and Z shell, respectively, to signify that the shell is waiting for you to type a new command.

Now, try your first command. Type **ls** in your terminal and press Enter. The command **ls** means "list". You use it to list files and directories. You will learn a lot more about these terms in the [next section](#). If you're on macOS, you will see something similar to this.

```
appacademy@demo ~ % ls
Desktop  Downloads  Movies    Pictures
Documents Library    Music     Public
appacademy@demo ~ %
```

The command **ls** has printed out eight different things that exist in your home folder! Then, another prompt shows up waiting for you to type something else.

On Ubuntu for Windows, you will see something like this.

```
appacademy@DESKTOP-MUCK09F:~$ ls
appacademy@DESKTOP-MUCK09F:~$
```

This is what would happen if your directory contains nothing. The command just prints nothing. Then, the shell prompts you for another command.

In these instructions, you should always press Enter after typing in a command. The instructions will not always specify that you should press Enter after typing your command. It's up to you to do that.

Bad commands

If you type something that the shell doesn't know how to run, you will get an error message. For example, try typing a command named **not-found** and pressing Enter.

On Ubuntu for Windows, you will see something like this.

```
appacademy@DESKTOP-MUCK09F:~$ not-found
not-found: command not found
appacademy@DESKTOP-MUCK09F:~$
```

On macOS, you will see something like this.

```
appacademy@demo ~ % not-found
zsh: command not found: not-found
appacademy@demo ~ %
```

It's important to carefully read error messages. They often contain good information and suggestions to help you type the correct command.

Clearing your shell

Your shell can get full of information and commands. It may get a little overwhelming. You can type the command `clear` and press Enter to clear the content of the shell.

Moving Around Directories

🕒 10 minutes

✅ Completed

Moving Around Directories

You're going to want to learn how to move around directories so that you can find and create files, run your programs, and do amazing stuff. This section is your introduction to those mechanics.

Your file system and you

The part of the operating system responsible for managing files and directories is called the **file system**. Files contain data. The file system organizes and keeps track of your files. Files are then stored in directories. Directories can hold both files *and* other directories.

Sometimes directories are called "folders". That makes sense because most operating systems like Windows and macOS use a picture of a folder as a symbol for a directory. This content will always try to call them directories. Don't let it confuse you. They're synonyms when it comes to computers.

As programmers, we will use the same small number of commands to create, inspect, rename, and delete files and directories. A very useful command to learn is the one that tells you where you are in the file system. The command

is `pwd` which stands for "present working directory". Open a terminal and type `pwd`.

In all of these examples, the name "appacademy" is the name of the user typing these commands. If your user name is "carla", then you will see "carla" in the place where the word "appacademy" exists in the examples.

Learning to generalize from examples is an important skill to acquire. This will let you get the most out of the different resources found on the Internet.

You should see something similar to `/home/appacademy` (Ubuntu for Windows) or `/Users/appacademy` (macOS). This is your **home directory**. In this directory, you should put all of your files organized into meaningful directories. That way you can keep track of where everything is.

You will now change the present working directory to another directory. To do that, you will use the `cd` command which stands for "change directory". To use the "change directory" command, you type "cd" and the **path** to the directory that you want to go to.

To change the present working directory to the top-most directory, known as the **root directory**, type `cd /`. That means "change directory to the directory named '/". Because the directory path that you're changing to begins with "/", that is an **absolute path**. That interaction should have looked something like this. (Again, remember, your name and the computer name and the prompt symbol may be different. Try to *generalize* what you see so that you can apply it to your specific shell output.) The first output is Ubuntu for Windows. The second is macOS.

```
appacademy@DESKTOP-MUCK09F:~$ cd /
appacademy@DESKTOP-MUCK09F:/$
```

```
appacademy@demo ~ % cd /
appacademy@demo / %
```

Notice that the directory portion of the prompt changed from "~" (which is a nickname for your "home directory") to "/" (which is the name of the "root directory").

Now, type the command that lists the files and directories in the present working directory. (You learned this command in the last reading.) Again, the first output is what you'd see on Ubuntu for Windows. The second is what you'd see on macOS.

```
appacademy@DESKTOP-MUCK09F:/$ ls
bin    boot  dev  etc  home  init  lib
lib64  media mnt  opt  proc  root  run
sbin   snap  srv  sys  tmp   usr   var
appacademy@DESKTOP-MUCK09F:/$
```

```
appacademy@demo / % ls
Applications  Volumes  etc      sbin
Library       bin      home     tmp
System        cores   opt      usr
Users         dev     private  var
appacademy@demo / %
```

You can see that there are some similarly named entries between these two systems like "bin", "dev", "etc", and "var". That's because macOS and Ubuntu are both called "Unix-like" systems. All Unix-like systems have similarities like this. That helps you as a programmer be able to work on a variety of operating systems (like macOS and various versions of GNU/Linux of which Ubuntu is one) without having to learn all brand new commands. You will find the commands `pwd`, `ls`, and `cd` on all of the Unix-like systems. You will also find common directories that contain similar types of files.

Absolute directory paths

The top-most directory, the **root directory**, it has the name "/". This is an abbreviated output of the entries in the Ubuntu for Windows root directory.

```
/
├─ bin/
├─ boot/
├─ dev/
├─ etc/
├─ home/
├─ init*
├─ lib/
├─ ... lots more entries here ...
├─ sys/
├─ tmp/
├─ usr/
└─ var/
```

The last entry is "var" which appears to be a directory because of the appended "/". That means that "var" is a **subdirectory** of the root directory. You would write the **absolute path** for "var" starting with the root path "/" and adding the "var/" path to it to get `/var/`. Thus, `/var/` is the absolute path to the subdirectory named "var" under the root directory.

Inside `/var/`, there may be these entries.

```
/var
├─ backups/
├─ cache/
├─ crash/
├─ lib/
├─ local/
├─ lock -> /run/lock/
├─ log/
└─ mail/
```

```
|— opt/  
|— run -> /run/  
|— snap/  
|— spool/  
|— tmp/
```

The last of those is "tmp" which also appears to be a directory because of the appended "/". That means that "tmp" is a subdirectory of "var". You would write the absolute path for "tmp" starting with the root path "/" and adding the "var/" path to it and, then, adding the "tmp/" path to it to get `/var/tmp/`.

"Flags" for commands

You can tell commands to do different things by passing them different instructions. For example, when you run `ls`, it just shows you the names of the files and directories. You can tell `ls` to get more fancy. Type `ls -F`, now. Look at the difference of the output. All of the names that are directories contained in the present working directory now have a slash after their names. If you have any files that are executable (that you can run them), then those files have an asterisk "*" after their names. On Ubuntu for Windows, here's what the output looks like.

```
appacademy@DESKTOP-MUCK09F:/$ ls -F  
bin/   boot/   dev/   etc/   home/  init*  lib/  
lib64/ media/  mnt/   opt/   proc/  root/  run/  
sbin/  snap/   srv/   sys/   tmp/   usr/   var/  
appacademy@DESKTOP-MUCK09F:/$
```

That shows 20 directories and one executable file.

Any extra instructions that you give to a command that begins with a dash like "-F" is called a **flag**. The most common flags for `ls` are

- `-a` which means do not ignore entries that begin with "."
- `-l` which tells the command to show lots of information about each entry
- `-F` which appends an indicator to inform you of the kind of entry it is

You can use them all in the same command, too!

```
appacademy@DESKTOP-MUCK09F:/$ ls -alF  
total 580  
drwxr-xr-x  1 root root    512 Dec 12 10:51 ./  
drwxr-xr-x  1 root root    512 Dec 12 10:51 ../  
drwxr-xr-x  1 root root    512 Dec 12 10:59 bin/  
drwxr-xr-x  1 root root    512 May 21  2019 boot/  
drwxr-xr-x  1 root root    512 Dec 13 03:51 dev/  
drwxr-xr-x  1 root root    512 Dec 13 03:51 etc/  
drwxr-xr-x  1 root root    512 Dec 12 10:52 home/  
-rwxr-xr-x  1 root root 591344 Dec 31  1969 init*  
drwxr-xr-x  1 root root    512 May 21  2019 lib/  
drwxr-xr-x  1 root root    512 May 21  2019 lib64/  
drwxr-xr-x  1 root root    512 May 21  2019 media/  
drwxr-xr-x  1 root root    512 Dec 12 10:51 mnt/  
drwxr-xr-x  1 root root    512 May 21  2019 opt/  
dr-xr-xr-x  9 root root      0 Dec 13 03:51 proc/  
drwx----- 1 root root    512 May 21  2019 root/  
drwxr-xr-x  1 root root    512 Dec 13 03:51 run/  
drwxr-xr-x  1 root root    512 Dec 12 11:01/sbin/  
drwxr-xr-x  1 root root    512 Mar 21  2019 snap/  
drwxr-xr-x  1 root root    512 May 21  2019 srv/  
dr-xr-xr-x 12 root root      0 Dec 13 03:51 sys/  
drwxrwxrwt  1 root root    512 Dec 13 04:46 tmp/  
drwxr-xr-x  1 root root    512 May 21  2019 usr/  
drwxr-xr-x  1 root root    512 May 21  2019 var/  
appacademy@DESKTOP-MUCK09F:/$
```

"Arguments" for commands

Rather than modifying the way it works, you can give many commands **arguments** which specify files or directories on which the command should operate. You've already done this back when you changed directory from your home directory to the root directory. You wrote `cd /`. The "/" is the argument for the "cd" command.

When you type `ls`, you're telling the list command to list the contents of the present working directory. You can also specify an argument to the list command to tell it to list the contents of a specific directory! Try typing the command `ls -alF /bin` and get ready for lots of output! You have told the list command to show "all" files in "long" format with "Fancy" indicators for the entry named "bin" in the root directory "/". When you specify the path that you want the list of entries, that means you can do that from *any* present working directory.

Try typing `ls -alF /var/`. You can do that from any directory. It will always show the contents of the "var" subdirectory of the root directory because you have provided that argument to the list command.

Now, type `ls -alF /bin/ls`. You should see something similar to the following.

```
appacademy@DESKTOP-MUCK09F:/$ ls -alF /bin/ls
-rwxr-xr-x 1 root root 133792 Jan 18  2018 /bin/ls*
```

That's an "executable" file. You can tell that because of the "*" appended to the name. What's neat is that `/bin/ls` is the executable file that the operating system runs every time you've been typing `ls`! How cool is that?

Relative paths

You've figured out the absolute path thing. Absolute paths begin with the "/" character and have names of directories and files after that, each one separated by another "/". Like, the list command can be found at `/bin/ls` and the temporary files the computer needs to run are stored in `/var/tmp`. If you want to move between directories but don't want to specify the absolute full path, then you can use a **relative path** to specify the argument.

Make sure that you're in the root directory by typing `cd /`. Your prompt should have the "/" character in it. If you type the `pwd` command, it should report "/".

Now, you can change the present working directory to the "var" subdirectory by typing the absolute path `cd /var/`. You can also change to the "var" subdirectory by just typing the name of the directory that you want to change into `cd var`. Try typing `cd var` into your terminal followed by a `pwd` command.

```
appacademy@DESKTOP-MUCK09F:/$ cd var
appacademy@DESKTOP-MUCK09F:/var$ pwd
/var
appacademy@DESKTOP-MUCK09F:/var$
```

You can see that the present working directory is now "/var".

Now, if you wanted to change to the "tmp" subdirectory of "var", you could type the absolute path to "tmp" like this: `cd /var/tmp/`. However, that's a lot of typing and, as the paths get longer and longer, that's more for you to have to remember. Luckily, you can just use a relative path! Because your terminal is presently working in the "/var" directory, you can change to the "tmp" subdirectory merely by typing the subdirectory's name, saving a lot of typing and headaches.

```
appacademy@DESKTOP-MUCK09F:/var$ cd tmp
appacademy@DESKTOP-MUCK09F:/var/tmp$ pwd
/var/tmp
appacademy@DESKTOP-MUCK09F:/var/tmp$
```

A special relative path is the path `..` (pronounced *dot dot*). It is a special directory name meaning “the directory containing this one”, or more succinctly, the parent of the current directory. When you type `cd ..`, it will move you from the current directory to its parent directory.

The present working directory for your shell is `/var/tmp`. That means that you're in the "tmp" subdirectory. You can move to its parent directory, "var", by typing `cd ..`.

```
appacademy@DESKTOP-MUCK09F:/var/tmp$ cd ..
appacademy@DESKTOP-MUCK09F:/var$ pwd
/var
appacademy@DESKTOP-MUCK09F:/var$
```

Another special relative path is the path `~` which means "the current user's home directory". So, instead of typing `cd /home/appacademy` or `cd /Users/appacademy` to return to a home directory, you can just type `cd ~` and it will change the present working directory from wherever you are to your home directory. Try it.

```
appacademy@DESKTOP-MUCK09F:/var$ cd ~
appacademy@DESKTOP-MUCK09F:~$ pwd
/home/appacademy
appacademy@DESKTOP-MUCK09F:~$
```

As a matter of fact, it's such a common thing to want to return to your home directory, that when you type the "change directory" command with no

arguments it will take you to your home directory. So, instead of typing `cd ~`, you can just type `cd`! Yay, efficiency!

What you've learned

In this section, you learned quite a bit. Here are the highlights of the learning.

- The file system is responsible for managing information on the disk.
- Information is stored in files, which are stored in directories.
- Directories can also store other directories, which forms a "directory tree".
- `cd path` changes the present working directory to the indicated path.
- `ls path` prints a listing of a specific file or directory; `ls` on its own lists the contents of the present working directory.
- `pwd` prints the user's present working directory.
- `/` on its own is the *root directory* of the whole file system.
- A *relative path* specifies a location starting from the current location.
- An *absolute path* specifies a location from the root of the file system.
- Directory names in a path are separated with `/` on Unix-like systems.
- `..` means ‘the directory above the current one’.

Working With Files and Directories

🕒 10 minutes

✅ Completed

Working With Files And Directories

Now that you can move about directories and list their contents, it's time for you to learn to create, copy, and delete files and directories.

Creating directories

You'll start by creating a new directory named "prep-work" in your home directory. Make sure you're in your home directory by typing `cd ~` or `cd` and pressing enter in your terminal.

To create a directory, you will use the "make directory" command which is named "mkdir" and takes an argument that is the name of the directory that you want to create. You want to create a directory named "prep-work" in your home directory, so you will type `mkdir prep-work`. After you've done that, make sure it exists by using the list command (`ls`) to see the entries in your home directory.

```
appacademy@DESKTOP-MUCK09F:/$ cd ~
appacademy@DESKTOP-MUCK09F:~$ mkdir prep-work
```

```
appacademy@DESKTOP-MUCK09F:~$ ls -F
prep-work/
appacademy@DESKTOP-MUCK09F:~$
```

What do you think would happen if you tried to do that again? What do you think should happen if you tried to create a directory with a name of an already existing directory? Go ahead and try it.

```
appacademy@DESKTOP-MUCK09F:~$ mkdir prep-work
mkdir: cannot create directory 'prep-work': File exists
appacademy@DESKTOP-MUCK09F:~$
```

There is the `mkdir` command giving you an error message that it can't create that directory because something already exists with that name.

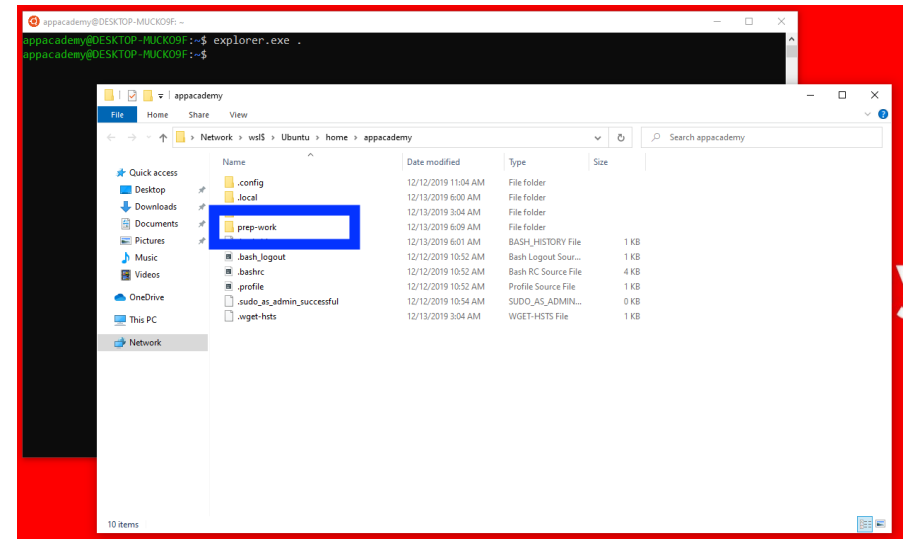
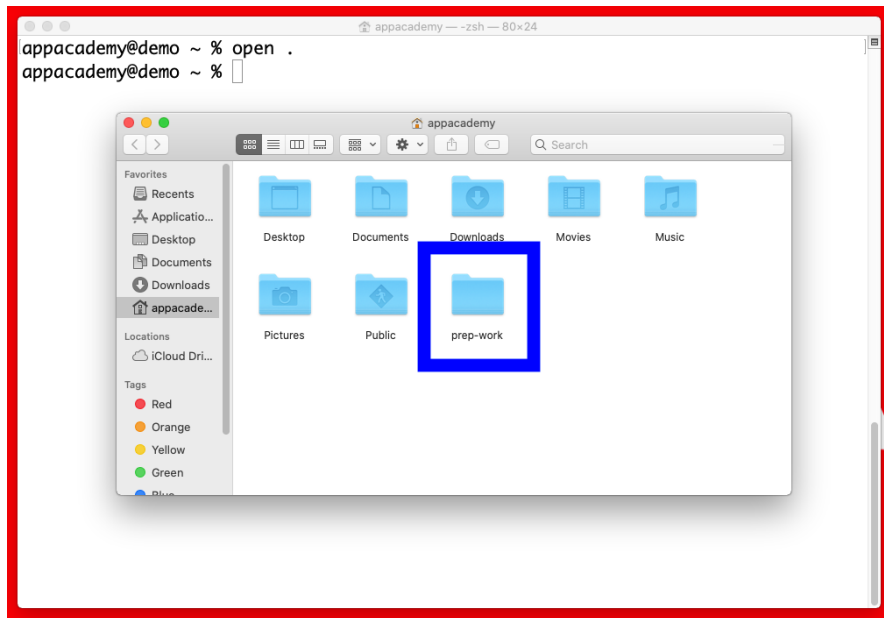
In Unix-like systems, the word "file" has two meanings. It can mean the collection of data that is some information, like an image or a spreadsheet. It also is a generic term for any entry in the "file system". You see, it's called a "file system", not a "file and directory system". So, in this case, when you try to create a directory that already exists, the command is using the second definition of "file" meaning something in the file system.

Using the shell to create a directory is no different than using a file explorer. That's called "Explorer" in Windows and "Finder" on macOS. If you open the current directory using your operating system's graphical file explorer, the "prep-work" directory will appear there too. While the shell and the file explorer are two different ways of interacting with the files, the files and directories themselves are the same. To see this, you can start the file explorer from the command line.

- For macOS, type `open .`
- For Ubuntu for Windows, type `explorer.exe .` (If for some reason this does not work, try closing your terminal and opening it again. If that still

does not work, try rebooting Windows.)

These will open the graphical file explorers. You should be able to see your newly-created directory named "prep-work" in them.



Good names for files and directories

Complicated names of files and directories can make your life painful when working on the command line. Here are a few useful tips for the names of your files.

- **Don't use spaces:** Spaces can make a name more meaningful, but since spaces are used to separate arguments on the command line it is better to avoid them in names of files and directories. You can use - or _ instead (e.g. north-pacific-gyre/ rather than north pacific gyre/).
- **Don't begin the name with - (dash):** Commands treat names starting with - as options.
- Stick with letters, numbers, . (period or 'full stop'), - (dash) and _ (underscore).

Many other characters have special meanings on the command line. You will learn about some of these during this prep work. There are special characters that can cause your command to not work as expected and can even result in data loss.

If you need to refer to names of files or directories that have spaces or other special characters, you should surround the name in quotes ("").

Create a text file

In the terminal, change the present working directory to the new "prep-work" directory. Make sure you're there by using the `pwd` command. Now, type `code notes.txt`. This should open Visual Studio Code with an empty tab with the name "notes.txt" in it.

Take a moment and type some notes in it about what you've learned so far. Don't just type some nonsense or nothing at all. That will cheat you out of a chance to perform some recall, to help you build "durable knowledge". That's the best kind of knowledge. Try typing the commands that you know and what each of them does. Write an explanation of what a "home directory" is. Once you're done, save your file (File > Save in the menu, or Command+S on a Mac or Ctrl+S on Windows). Now, exit Visual Studio Code (File > Exit on Windows or Code > Quit Visual Studio Code on Mac).

Check to see that the file exists by listing the contents of your present working directory. If you don't see it, make sure that you are in the correct directory (/home/«your name here»/prep-work) and that you did save your file.

Moving files and directories

Your file is named "notes.txt". Now, you realize that you want to name it "my-notes.txt" because you might share notes with someone else. Renaming a file from the command line is moving the file from one name to another. You use the `mv` command which means "move". It takes two arguments: the file that you want to move and where you want to move it.

```
appacademy@DESKTOP-MUCK09F:~/prep-work$ ls
notes.txt
appacademy@DESKTOP-MUCK09F:~/prep-work$ mv notes.txt my-notes.txt
appacademy@DESKTOP-MUCK09F:~/prep-work$ ls
my-notes.txt
appacademy@DESKTOP-MUCK09F:~/prep-work$
```

Notice the use of *relative paths* as the arguments of the `mv` command. Because it's just the file names, the `mv` command ends up looking in the present working directory for the files.

What if you want to move your notes out of the "prep-work" directory? You want to create a "notes" directory in your home directory and move "my-notes.txt" into the new "notes" directory *without changing the present working directory!* You can do that.

This will be a two step process. Think about how you can use the home directory shortcut "~" and relative paths.

1. Create a new directory named "notes" as a subdirectory of your home directory.
2. Move the file from this location to the newly-created "notes" directory.

Try doing it yourself before going on.

The following code checks to make sure that "my-notes.txt" is in the current directory, makes the new "notes" directory in your home directory, moves the "my-notes.txt" file to the newly-created directory, checks that the current directory no longer has the "my-notes.txt" file in it, and confirms that it now exists in the newly-created "notes" directory.

```
appacademy@DESKTOP-MUCK09F:~/prep-work$ ls
my-notes.txt
appacademy@DESKTOP-MUCK09F:~/prep-work$ mkdir ~/notes
appacademy@DESKTOP-MUCK09F:~/prep-work$ mv my-notes.txt ~/notes/
appacademy@DESKTOP-MUCK09F:~/prep-work$ ls
appacademy@DESKTOP-MUCK09F:~/prep-work$ ls ~/notes
my-notes.txt
appacademy@DESKTOP-MUCK09F:~/prep-work$
```

You could also use `mv my-notes.txt ../notes` instead of `mv my-notes.txt ~/notes/`. Why?

Copying files and directories

The `cp` command works very much like `mv`, except it copies a file instead of moving it.

Removing files

The command to remove a file is `rm`, short for "remove".

The shell doesn't have a trash bin that you can recover deleted files from (though most graphical interfaces do). Instead, when you delete files, they are unlinked from the file system so that their storage space on disk can be

recycled. So, if you `rm`, it's gone forever. That's why it's a good idea to always use the `-i` flag to have it prompt you to confirm that you really do want to remove the file.

Wildcards

"*" is a wildcard, which matches zero or more characters. Consider the "notes" directory: *.txt matches "my-notes.txt" and "carlas-notes.txt" and "micheles-notes.txt" but *not* "mikes-notes.doc". On the other hand, "m*.txt" only matches "my-notes.txt" and "micheles-notes.txt", because the 'm' at the front only matches filenames that begin with the letter 'm'.

"?" is also a wildcard, but it matches exactly one character. So "?y-notes.txt" would match "my-notes.txt".

Wildcards can be used in combination with each other e.g. "????otes.txt" matches four characters followed by "otes.txt", giving "my-notes.txt".

When the shell sees a wildcard, it expands the wildcard to create a list of matching filenames before running the command that was asked for. As an exception, if a wildcard expression does not match any file, Bash will pass the expression as an argument to the command as it is. For example typing *ls .pdf in the molecules directory (which contains only files with names ending with .pdf) results in an error message that there is no file called .pdf.* However, generally commands like `wc` and `ls` see the lists of file names matching these expressions, but not the wildcards themselves. It is the shell, not the other programs, that deals with expanding wildcards, and this is another example of orthogonal design.

An exercise

You're starting a new class, and would like to duplicate the directory structure from your previous class so you can add new notes and homework.

Assume that the previous class is in a folder called "french-cooking", which contains a "homework" folder that in turn contains folders named "recipes" and "menus" that contain text files. The goal is to reproduce the folder structure of the "french-cooking/data" folder into a folder called "british-cooking" so that your final directory structure looks like this:

```
~
└─ british-cooking/
    └─ homework/
        ├── menus/
        └─ recipes/
```

Starting in your home directory, what commands could you type to get those directories like that?

What you've learned

You've learned quite a bit more. Here's a review of those topics to make sure that you got everything.

- `cp` copies a file.
- `mkdir my-new-dir` creates a new directory named "my-new-dir".
- `mv` moves (renames) a file or directory.
- `rm my-file` removes (deletes) a file named "my-file".
- The wildcard "*" matches zero or more characters in a filename, so "*.txt" matches all files ending in ".txt".

- The wildcard "?" matches any single character in a filename, so "?.txt" matches "a.txt" but not "any.txt".
- The shell does not have a trash bin: once something is deleted, it's really gone.
- Most file names are "something.extension". The extension isn't required, and doesn't guarantee anything, but is normally used to indicate the type of data in the file.

Introduction to HTML

🕒 30 minutes

✅ Completed

HTML (Hypertext Markup Language) is the most basic of all things that you must learn to make something visual for the World Wide Web. HTML is a very easy language to learn and super simple to write. You start off with

```
<html>
  <head>
    <title>An HTML Document with No Body</title>
  </head>
  <body>
</body>
</html>
```

As you can see, the HTML language uses words contained inside angle brackets: `<` and `>`, known as tags or elements. You also might notice that every tag is almost duplicated, but the second tag has a `/` before it. **Open tags** are the ones without the slashes, like `<html>` and `<body>`. **Close tags** are the ones with the slashes, like `</html>` and `</body>`. Everything between an open tag and its corresponding close tag is the **tag content**.

In the above HTML snippet, the `html` tag wraps the entire document. All other content should be within the `html` open and close tags.

The `head` tag contains metadata about the document. In this case, it contains the `title` tag which specifies the title of the HTML document which, in this

case, is "An HTML Document with No Body".

Remember, HTML was invented so scientists could share their results. Phrases like "HTML document" hearken back to that time when the "document" was a research paper and HTML was the format it was written in.

Between the open and close `body` tag should be the place where you put all of the visible content of your HTML document.

The phrase **HTML element** is sometimes used interchangeably with the concept of the everything from the open tag, all the content, all the way to the close tag.

Some elements have *no* close tag. These are called "empty" in that they have no content.

Attributes

All HTML elements can have "attributes". These add more information to the HTML element. The three attributes in the following list can be applied to any HTML element.

- **id** – gives each element a unique identifier to be selected with CSS
- **class**– assigns an element to a class to be selected with CSS or JavaScript
- **title**– gives a cool hover feature that displays text over an element

Here's an example of those attributes on a paragraph tag.

```
<p id="paragraph-1" class="fancy-paragraph" title="I am a title">
  Hover over me to see the title
```


</p>

Each value for the "id" attribute must be unique across the *entire* HTML document. That gives each element its own unique identifier that you can use as a CSS selector to apply styling to one specific element.

The "class" attribute is another way for CSS to be able to select HTML elements. Classes are meant to be applied to more than one element. That means that there's nothing wrong with the following HTML even though the paragraph tags have the same class value. (Note that the ids are different, though.)

```
<p id="paragraph-1" class="fancy-paragraph" title="I am a title">
  Hover over me to see the title
</p>
<p id="paragraph-2" class="fancy-paragraph">
  I have not title. I am sad.
</p>
```

If an HTML element needs to have more than one class, you write *space-delimited list* in the class attribute's value like this.

```
<p id="paragraph-1" class="fancy-paragraph error-message">
  Hover over me to see the title
</p>
```

This way the paragraph would have the CSS rules applied to it for all paragraph tags, any rules for the class "fancy-paragraph", any rules for the class "error-message", and any rules for the HTML element with the id of "paragraph-1".

Hyperlink

In itself, an HTML hyperlink is one of the most useful distinctions between the World Wide Web and a standard text file. Linking is the foundation of the World Wide Web. You can employ links to provide navigation to other content. Remember that hyperlinks are created in HTML with **anchor tags**, that is, the `<a>` tag.

Relative and absolute links

Absolute links are generally used to send readers of your Web site to another Web site. For example, <https://appacademy.io> is an absolute link because it starts with "http://" or "https://".

Relative links do not start with "http://" or "https://". Instead, they're like relative paths with respect to the file system.

```
<a href="https://appacademy.io"> Absolute Link</a>
<a href="subdirectory/another-page.html"> Relative Link</a>
```

Headings

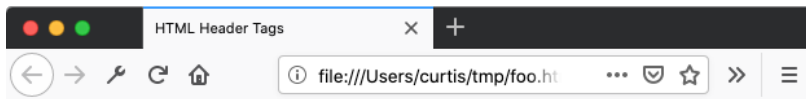
Remember that HTML was originally designed to let scientists publish their results so that others could read them on their computers? Well, any good scientific paper must have headers. You have them in Microsoft Word and Google Docs. You might as well have them in HTML, too!

There are six levels of headers, each represented by a different tag: `h1`, `h2`, `h3`, `h4`, `h5`, and `h6`.

```

<html>
  <head>
    <title>HTML Head Tag - Tutorial and Reference</title>
  </head>
  <body>
    <h1>I am very very important.</h1>
    <h2>I am very important.</h2>
    <h3>I am important.</h3>
    <h4>I am kind of important.</h4>
    <h5>I am somewhere below important.</h5>
    <h6>I am near the importance and size of a paragraph tag.</h6>
  </body>
</html>

```



I am very very important.

I am very important.

I am important.

I am kind of important.

I am somewhere below important.

I am near the importance and size of a paragraph tag.

A nice rule to observe is that you should appropriately stack your headings. You should never have an `h3` directly following an `h1`. While it doesn't break the HTML, it defeats the intent of the *structure* of the document. You shouldn't want to have a hole in your hierarchy.

Visually impaired users that interact with the World Wide Web with a screen reader cannot see the tag following the “hole”. Always structure your tags to have the next larger tag before it.

Div and span tags

Div tags and span tags are very common HTML elements used to create additional structure for the use of styling or grouping. Because `div` tags are "block" tags, they were primarily used to create sections on a Web page before HTML 5 came along and introduced the `section` tag.

Span tags primarily used to style text within a larger body of text.

For example, in the code blocks on this page, the code block itself is a `div`. Each differently-colored element is wrapped in a `span` tag.

Image tag

Unlike most elements, the image tag allows you to show an image in your Web page. The HTML image tag instructs the browser to retrieve the image and displays it in your Web page. Like an anchor tag which needs an `href` attribute to know where to send the user when they click on the link, the image tag needs a `src` attribute to provide the URL (absolute or relative) of where its located. Image tags are *empty tags* in that they have no content and, therefore, no end tag.

```

```

A required attribute for the image tag is the "alt" attribute. The value of the "alt" attribute is displayed if an image fails to load. Also, visually impaired people using a screen reader will hear the "alt" text as a description of the image.

```
&lt;html&gt;</code> element represents the root (top-level element) of an HTML document, so it is also referred to as the root element. All other elements must be descendants of this element. |
| head    | The HTML <code>&lt;head&gt;</code> element contains machine-readable information (metadata) about the document, like its title, scripts, and style sheets.                                                     |
| body    | The HTML <code>&lt;body&gt;</code> Element represents the content of an HTML document. There can be only one <code>&lt;body&gt;</code> element in a document.                                                  |

### Document metadata

| Element | Description                                                                                                                                                                                                                                                                                                                                                        |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| link    | The HTML External Resource Link element ( <code>&lt;link&gt;</code> ) specifies relationships between the current document and an external resource. This element is most commonly used to link to stylesheets, but is also used to establish site icons (both "favicon" style icons and icons for the home screen and apps on mobile devices) among other things. |
| title   | The HTML Title element ( <code>&lt;title&gt;</code> ) defines the document's title that is shown in a browser's title bar or a page's tab.                                                                                                                                                                                                                         |

### Content sections

| Element      | Description                                                                                                                                                                                                                                                                                       |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| footer       | The HTML <code>&lt;footer&gt;</code> element represents a footer for its nearest sectioning content or sectioning root element. A footer typically contains information about the author of the section, copyright data or links to related documents.                                            |
| header       | The HTML <code>&lt;header&gt;</code> element represents introductory content, typically a group of introductory or navigational aids. It may contain some heading elements but also a logo, a search form, an author name, and other elements.                                                    |
| h1, h2, etc. | The HTML <code>&lt;h1&gt;</code> – <code>&lt;h6&gt;</code> elements represent six levels of section headings. <code>&lt;h1&gt;</code> is the highest section level and <code>&lt;h6&gt;</code> is the lowest.                                                                                     |
| main         | The HTML <code>&lt;main&gt;</code> element represents the dominant content of the <code>&lt;body&gt;</code> of a document. The main content area consists of content that is directly related to or expands upon the central topic of a document, or the central functionality of an application. |
| nav          | The HTML <code>&lt;nav&gt;</code> element represents a section of a page whose purpose is to provide navigation links, either within the current document or to other documents. Common examples of navigation sections are menus, tables of contents, and indexes.                               |
| section      | The HTML <code>&lt;section&gt;</code> element represents a standalone section — which doesn't have a more specific semantic element to represent it — contained within an HTML document.                                                                                                          |

### Text content

| Element | Description                                                                                                                                                         |
|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| div     | The HTML Content Division element ( <code>div</code> ) is the generic container for flow content. It has no effect on the content or layout until styled using CSS. |
| li      | The HTML <code>li</code> element is used to represent an item in a list.                                                                                            |
| ol      | The HTML <code>ol</code> element represents an ordered list of items — typically rendered as a numbered list.                                                       |
| p       | The HTML <code>p</code> element represents a paragraph.                                                                                                             |
| ul      | The HTML <code>ul</code> element represents an unordered list of items, typically rendered as a bulleted list.                                                      |

## Inline text semantics

| Element | Description                                                                                                                                                                                                                                                                          |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| a       | The HTML <code>a</code> element (or anchor element), with its href attribute, creates a hyperlink to web pages, files, email addresses, locations in the same page, or anything else a URL can address.                                                                              |
| br      | The HTML <code>br</code> element produces a line break in text (carriage-return). It is useful for writing a poem or an address, where the division of lines is significant.                                                                                                         |
| em      | The HTML <code>em</code> element marks text that has stress emphasis. The <code>em</code> element can be nested, with each level of nesting indicating a greater degree of emphasis.                                                                                                 |
| span    | The HTML <code>span</code> element is a generic inline container for phrasing content, which does not inherently represent anything. It can be used to group elements for styling purposes (using the class or id attributes), or because they share attribute values, such as lang. |
| strong  | The HTML Strong Importance Element ( <code>strong</code> ) indicates that its contents have strong importance, seriousness, or urgency. Browsers typically render the contents in bold type.                                                                                         |

## Image and multimedia

| Element | Description                                                          |
|---------|----------------------------------------------------------------------|
| img     | The HTML <code>img</code> element embeds an image into the document. |

## Table content

| Element | Description                                                                                                                                                                                        |
|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| table   | The HTML <code>table</code> element represents tabular data — that is, information presented in a two-dimensional table comprised of rows and columns of cells containing data.                    |
| tbody   | The HTML Table Body element ( <code>tbody</code> ) encapsulates a set of table rows ( <code>tr</code> elements), indicating that they comprise the body of the table ( <code>table</code> ).       |
| td      | The HTML <code>td</code> element defines a cell of a table that contains data. It participates in the table model.                                                                                 |
| tfoot   | The HTML <code>tfoot</code> element defines a set of rows summarizing the columns of the table.                                                                                                    |
| th      | The HTML <code>th</code> element defines a cell as header of a group of table cells. The exact nature of this group is defined by the scope and headers attributes.                                |
| thead   | The HTML <code>thead</code> element defines a set of rows defining the head of the columns of the table.                                                                                           |
| tr      | The HTML <code>tr</code> element defines a row of cells in a table. The row's cells can then be established using a mix of <code>td</code> (data cell) and <code>th</code> (header cell) elements. |

## Forms

| Element  | Description                                                                                                                                                                |
|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| button   | The HTML <code>button</code> element represents a clickable button, which can be used in forms or anywhere in a document that needs simple, standard button functionality. |
| fieldset | The HTML <code>fieldset</code> element is used to group several controls as well as labels ( <code>label</code> ) within a web form.                                       |

| Element  | Description                                                                                                                                                                                                                                                                      |
|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| form     | The HTML <code>form</code> element represents a document section that contains interactive controls for submitting information to a web server.                                                                                                                                  |
| input    | The HTML <code>input</code> element is used to create interactive controls for web-based forms in order to accept data from the user; a wide variety of types of input data and control widgets are available, depending on the device and user agent.                           |
| label    | The HTML <code>label</code> element represents a caption for an item in a user interface.                                                                                                                                                                                        |
| legend   | The HTML <code>legend</code> element represents a caption for the content of its parent <code>fieldset</code> .                                                                                                                                                                  |
| option   | The HTML <code>option</code> element is used to define an item contained in a <code>select</code> , an <code>optgroup</code> , or a <code>datalist</code> element. As such, <code>option</code> can represent menu items in popups and other lists of items in an HTML document. |
| select   | The HTML <code>select</code> element represents a control that provides a menu of options                                                                                                                                                                                        |
| textarea | The HTML <code>textarea</code> element represents a multi-line plain-text editing control, useful when you want to allow users to enter a sizable amount of free-form text, for example a comment on a review or feedback form.                                                  |

# Building Forms in HTML

🕒 12 minutes

✅ Completed

## Building Forms In HTML

Eventually, you'll want to have people send information to our applications so that you can send them stuff or get their opinions. You will do this by creating an HTML form. In this, you will design a simple form, implement it using the right HTML form controls and other HTML elements, add some styling via CSS, and cover how data is sent to a server.

**IMPORTANT:** Do not copy and paste the code from this page into your text editor. Type the code. Typing the code will give you better time to think about what's happening. That way, you can reflect on it. That way you can build durable knowledge.

## Getting started

1. If it's not open, open up your terminal. Change the present working directory to your home directory.
2. Make a new directory in your home directory named "html-form".
3. Change the present working directory into "html-form".

4. Start Visual Studio Code by running the command `code .` from the terminal in the "html-form" directory.
5. Create a new file.
6. Type the following HTML code into it.
7. Save the file as "add-user.html".

The code to *TYPE* in "add-user.html" is:

```
<!DOCTYPE html>
<html lang="en-US">
 <head>
 <meta charset="utf-8">
 <title>Learning the Forms</title>
 </head>
 <body>
 <h1>Hello, Forms!</h1>
 </body>
</html>
```

Open this in a browser. If you're on macOS, type `open add-user.html`. That will open the HTML file in your default browser. If you're on Windows, type `explorer.exe` in your Ubuntu command line. That will open Windows File Explorer. Double-click the "index" file that you see. That should open your file in your default browser.

## What are HTML forms?

HTML Forms are one of the oldest types of interaction between a person and an application. Forms allow users to enter data. Then, your application can decide to send it somewhere or do something with it locally.

An HTML Form is made of one or more input element types. Those types can be buttons, checkboxes, drop-downs, multi-selects, multi-line text fields, radio buttons, and single-line text fields. The single-line text fields can even require data entered to be of a specific format or value. Each form element can and should have a corresponding label to describe what the form elements expects. This helps sighted and blind people to interact with forms.

The main difference between a HTML form and a regular HTML document is that most of the time, the data collected by the form is sent to a web server. In that case, you need to set up a web server to receive and process the data. You'll get to that in the actual software engineering class.

## The form's design

Before starting to code anything with a visual element, it's always better to step back and take the time to think about what it will look like, how people will use your stuff. Designing a quick mock-up will help you to define the right set of data you want to ask people. From an interactive experience point of view, it's important to remember that the bigger your form, the more you risk losing users. Keep it simple and stay focused: ask only for that data you absolutely need.

In this, you will build a simple "add a user" form. It will look like this.

```

 ADD USER

 FIRSTNAME
 []
 LAST NAME
 []
 EMAIL
 []
 ROLE
 [Admin ▼]
 EXPIRATION
 [mm/dd/yyyy]
 BIO
 []

```

This form has four single-line text, one drop-down, and one multi-line text elements.

## Implementing your form HTML

Ok, now we're ready to go to HTML and code our form. To build our contact form, we will use the following HTML elements: `<form>`, `<label>`, `<input>`, `<textarea>`, and `<button>`.

### The `<form>` element



All HTML forms start with a `<form>` element like this:

```
<form action="/form-handling-url" method="post">

</form>
```

This element formally defines a form. It's a container element like a `<div>` or `<p>` element, but it also supports some specific attributes to configure the way the form behaves. Technically, all of its attributes are optional. It's standard practice to always set at least the `action` attribute and the `method` attribute:

- The `action` attribute defines the location (URL) where the form's collected data should be sent when it is submitted.
- The `method` attribute defines which HTTP method to send the data with. Browsers support only two values for this attribute: "get" and "post". You will use "post" 99% of the time.

For now, add the above `<form>` element into the HTML body after the `<h1>` element that reads "Hello, Forms!".

## The `<label>`, `<input>`, and `<textarea>` elements

The add user form is not complex. The form contains five text fields and one drop-down, each with a `<label>`. The input field for the first name is a single-line text field. The input field for the last name is a single-line text field. The input field for the email is an input of type *email*, a special text field that accepts only email addresses. The input field for expiration is an input of type *date*, a special text field that accepts only dates. The input field for the

message is a multiline text field. The element for the role is a drop-down of predefined items.

In terms of HTML code we need something like the following to implement these form elements. Update your form code to look like the code below.

*Don't copy and paste this code. Type it out. Get used to typing HTML. Get used to indentation. Get used to the way that it feels to write code. This is one of the methods through which you learn. Don't cheat yourself.*

```
<form action="/form-handling-url" method="post">
 <fieldset>
 <legend>Add user</legend>

 <div>
 <label for="firstName">First name</label>
 <input type="text" id="firstName" name="first_name">
 </div>

 <div>
 <label for="lastName">Last name</label>
 <input type="text" id="lastName" name="last_name">
 </div>

 <div>
 <label for="email">Email</label>
 <input type="email" id="email" name="email_address">
 </div>

 <div>
 <label for="role">Role</label>
 <select id="role" name="user_role_name">
 <option value="admin">Admin</option>
 <option value="user">User</option>
 <option value="guest">Guest</option>
 </select>
 </div>
```

```

<div>
 <label for="expiration">Expiration</label>
 <input type="date" id="expiration" name="expiration">
</div>

<div>
 <label for="bio">Bio</label>
 <textarea id="bio" name="bio"></textarea>
</div>

<div>
 <button type="submit">Add this person</button>
</div>

</fieldset>
</form>

```

The `<div>` elements are there to provide some structure to your code, to group the `<label>` and inputs together. For usability and accessibility, each form element has an associated label. Each label uses the `for` attribute to refer to the value of the `id` attribute on its corresponding form element. This associates the label to the form control enabling touching and clicking the label to activate the corresponding form element.

On the `<input>` element, the most important attribute is the `type` attribute. The `type` attribute tells the browser what kind of input it expects. In your form, you have three different types of inputs: `text`, `email`, and `date`. This will constrain the input to valid values. You won't, for example, be able to enter "Umbrella stand" for the `<input>` with the `date` value for its `type` attribute. It is not valid HTML to have a closing "input" tag. You should never write `<input type="text"></input>`.

The `<select>` element contains zero or more `<option>` elements. Having zero `<option>` elements makes very little sense, but it would be valid HTML. The content between the `<option>` and the `</option>` is what the browser

shows in the drop-down. The option that the person selects, the contents of its `value` attribute becomes the value of the `<select>`. So, if a person chose "Guest" in the drop-down, the value of the `<select>` would be "guest" because that's the content of the `value` attribute of the `<option>` with the text "Guest". The `<select>` element should have a closing `</select>`.

A `<textarea>` is a multi-line text input. It must have a closing tag like `</textarea>`.

Finally, the `<button>` element represents a button for the form. By default, if a button is inside a form, that is between a `<form>` and a `</form>` tag, then it is a *submit* button. That means that the form will send the content of each of the form fields to a server for processing. The content between the button tag and its closing tag `</button>` is the content that will appear on the button's face.

A completely unstyled form looks like this.

## Styling Your Form

Create a new file in Visual Studio Code and save it as "style.css" in the same directory as the "add-user.html" file above. In the `<head>` element, add the following HTML code.

```
<link rel="stylesheet" href="style.css">
```

The `link` element tells the browser that it wants to link the specified file to this specific page. The `rel` attribute reads "stylesheet" which means that you want the rules in that stylesheet to apply to this page. Finally, the `href` attribute contains a URL to the file that you want it to use. This is a *relative* URL since it does not begin with "http://" or "https://". It tells the browser to get the file in the same directory as the HTML file that contains this.

Styling HTML forms is hard. Type the following CSS into the *style.css* file. After every rule, save the *style.css* file and refresh the Web page to see the changes that it makes. Read each of the comments to understand how it affects the

```
fieldset {
 /* Get rid of the border on the fieldset. */
 border: 0;
}

form {
 /* Center the form on the page. */
 margin: 0 auto;
 width: 600px;

 /* Curved border around the form. */
 padding: 1em;
 border: 1px solid #CCC;
 border-radius: 1em;

 /* Make all the fonts in the form the same. */
 font-family: Arial, Helvetica, sans-serif;
}

/* Every div after the first div in the form. */
form div + div {
```

```
 margin-top: 1em;
}

label {
 /* Uniform size & alignment for labels. */
 display: inline-block;
 width: 90px;
 text-align: right;
}

legend {
 /* Add a bottom border to the legend. */
 border-bottom: 1px solid black;
}

input,
textarea {
 /* You must explicitly override the fonts in textareas and inputs. */
 font-family: Arial, Helvetica, sans-serif;

 /* Uniform text field and textarea sizes. */
 width: 425px;
 box-sizing: border-box;

 /* Match form field borders */
 border: 1px solid #999;
}

input:focus,
textarea:focus {
 /* Additional highlight for focused elements */
 border-color: #000;
}

textarea {
 /* Align multiline text fields with their labels */
 vertical-align: top;

 /* Provide space to type some text */
 height: 5em;
}
```

```

}

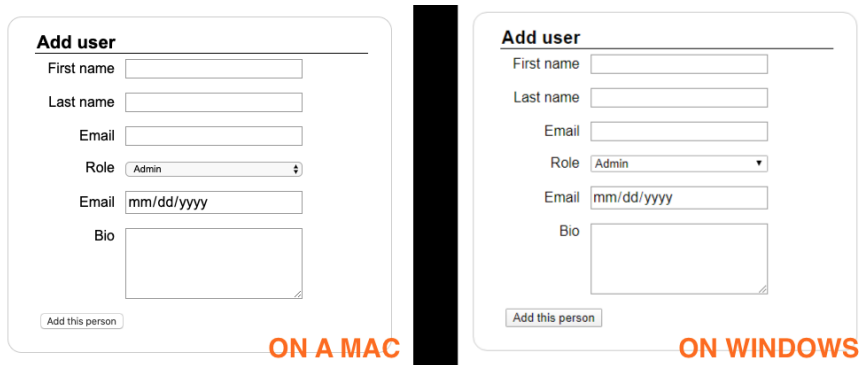
select {
 /* You must explicitly override the font in a select. */
 font-family: Arial, Helvetica, sans-serif;

 /* Make the select as wide as the other form elements. */
 width: 425px;
}

button {
 /* Space the button to the right. */
 margin-left: .5em;
}

```

Now that you've done that, your form should look like the following. Note how the styling allows the form to look nearly identical on Windows and macOS. This is the real strength of CSS. You can allow people using any kind of computer and device to experience your form in the way that makes sense to them.



## Media queries for narrow devices

The form is 600px wide no matter what and has padding of 16px all around it and a border of 1px all around it, which makes it about 635px wide. If the screen that the person is using to look at your form is less than 635px wide, they will have to scroll horizontally to use your form. That is considered bad design. To fix this, you can use a media query to adjust the way the form is laid out when the width is less than 650px, just to add some extra space in there to make sure everything appears to flow smoothly. Type the following at the end of your CSS file after the last rule (the `button` rule).

```

@media screen and (max-width: 650px) {

 /* remove the width from the form so that
 it is just as wide as the screen. */
 form {
 width: auto;
 }

 /* Turn labels into blocks so they can stretch
 across the entire screen. Also, get rid of
 the margin so that the text is left aligned.
 Finally, set the width to auto so that it
 isn't stuck at 90px. */
 label {
 display: block;
 margin: 0;
 text-align: left;
 width: auto;
 }

 /* Let the inputs, textareas, and selects span
 the entire screen, too. */
 input,
 select,
 textarea {
 width: 100%;
 }
}

```

Now, you can resize your screen and see the form change its layout based on the width of the device.

**Add user**

First name

Last name

Email

Role

Admin

Email

Bio

Add this person

# CSS Layout

🕒 14 minutes

✅ Completed

## Using CSS To Layout HTML Elements

This teaches the fundamentals of CSS that you will use to create any modern Web site. Now that you've become familiar with selectors, properties, and values, it's time to dive into all of the ways that you can use CSS to layout elements on a Web page.

### The "display" property

---

The "display" property is the most important property for controlling layout in CSS. Every element has a default display value. For most elements, that value is either "block" or "inline". In this material, "block" elements are sometimes referred to as "block-level" elements. You will come across both of those terms in many sources.

The `div` element is the standard block-level element. A block-level element starts on a new line and stretches out to the left and right as far as it can. Other common block-level elements are `p`, `form`, `header`, `footer`, and `section`.

The `span` element is the standard inline element. An inline element can wrap some text inside a paragraph without disrupting the flow of that paragraph.

The `a` element is the most common inline element that has extra functionality, since you use them for links.

Another common display value is "none". Some specialized elements such as `script` use this as their default. The `script` element is commonly used with JavaScript to hide and show elements without really deleting and recreating them. This is something that you will learn about in the online class.

A display value of "none" is different from visibility. Setting display to "none" will render the page as though the element does not exist. Setting visibility to "hidden" will hide the element, but the element will still take up the space it would if it was fully visible.

There are plenty of more exotic display values, such as "list-item" and "table" which are special types of display for list items and tables. Here is an [exhaustive list](#).

### The "margin" property set to "auto"

---

```
#main {
 width: 600px;
 margin: 0 auto;
}
```

Setting the width of a block-level element will prevent it from stretching out to the edges of its container to the left and right. Then, you can set the left and right margins to auto to horizontally center that element within its container. The element will take up the width you specify, then the remaining space will be split evenly between the two margins.

The only problem occurs when the browser window is narrower than the width of your element. The browser resolves this by creating a horizontal scrollbar on the page.

## Using "max-width" rather than "width"

---

```
#main {
 max-width: 600px;
 margin: 0 auto;
}
```

Using max-width instead of width in this situation will improve the browser's handling of small windows. This is important when making a site usable on mobile. Resize this page to check it out!

By the way, max-width is supported by all major browsers including IE7+ so you shouldn't be afraid of using it.

## Understanding the "box model" of layout

---

While diving into width, you should learn about width's big caveat: the box model. When you set the width of an element, the element can actually appear bigger than what you set it. Sometimes the element's border and padding will stretch out the element beyond the width that you specified. Look at the following example, where two elements with the same width value end up different sizes in the result.

```
.simple {
 width: 500px;
 margin: 20px auto;
```

```
}

.fancy {
 width: 500px;
 margin: 20px auto;
 padding: 50px;
 border-width: 10px;
}
```

For generations, the solution to this problem has been extra math. CSS authors have always just written a smaller width value than what they wanted, subtracting out the padding and border. This was hard. So a new CSS property called "box-sizing" was created. When you set "box-sizing" to the value of "border-box" on an element, the padding and border of that element no longer increase its width. Here are the improved selectors.

```
.simple {
 width: 500px;
 margin: 20px auto;
 box-sizing: border-box;
}

.fancy {
 width: 500px;
 margin: 20px auto;
 padding: 50px;
 border-width: 10px;
 box-sizing: border-box;
}
```

Many authors just set "border-box" on all elements on all their pages.

```
* { box-sizing: border-box; }
```

## Both yoga and CSS have positions

---

In order to make more complex layouts, you will need to use the "position" property. It has a bunch of possible values, and their names make no sense to people new to CSS which makes them hard to remember unless you use them.

```
.static {
 position: static;
}
```

For all HTML elements, "static" is the default value. An element with the CSS rule `position: static;` is not positioned in any special way and is said to be "unpositioned". If an element has any other value for its "position", then it is said to be "positioned".

```
.relative-like-static {
 position: relative;
}
```

By itself, "relative" acts just like "static". So, this does not really do anything.

```
.relative-and-obvious {
 position: relative;
 top: -20px;
 left: 20px;
}
```

Setting the "top", "right", "bottom", and "left" properties of a "relatively"-positioned element will cause it to be adjusted away from its normal position. Other content will not be adjusted to fit into any gap left by the element.

```
.fixed {
 position: fixed;
 bottom: 0;
 right: 0;
 width: 200px;
}
```

A "fixed" element is positioned relative to the viewport, which means it always stays in the same place even if the page is scrolled. As with relative, the "top", "right", "bottom", and "left" properties are used.

A "fixed" element does not leave a gap in the page where it would normally have been located.

```
.absolute {
 position: absolute;
 top: 120px;
 right: 0;
 width: 300px;
 height: 200px;
}
```

Elements with the "display" property set to "absolute" is the trickiest position value. The value "absolute" behaves like "fixed" except relative to the nearest positioned ancestor instead of relative to the viewport. If an absolutely-positioned element has no positioned ancestors, it uses the document body, and still moves along with page scrolling. Remember, a "positioned" element is one whose position is anything except static.

## Responding to different devices

---



"Responsive Design" is the strategy of making a site that "responds" to the browser and device that it is being shown on by changing the layout to adapt to the available space. Media queries are the most powerful tool for doing this.

```
@media screen and (min-width: 600px) {
 nav {
 position: absolute;
 top: 0;
 width: 100%;
 height: 50px;
 }
 main {
 margin-top: 50px;
 }
}

@media screen and (max-width: 599px) {
 nav {
 position: static;
 height: auto;
 }
}
```

When the width of the screen falls below 600px, then the nav changes to be inline rather than stuck to the top because it changes the "position" and "height" elements.

## Hybrid display "inline-block"

---

Web developers got sick of trying to hack the best of the "inline" and "block" display styles together. Elements with "inline-block" displays are like "inline" elements that can have widths and heights. You can use "inline-block" for total page layout.

- Elements with "inline-block" are affected by the "vertical-align" property, which you probably want set to top.
- You need to set the width of each column
- There will be a gap between the columns if there is any whitespace between them in the HTML

```
nav {
 display: inline-block;
 vertical-align: top;
 width: 25%;
}
```

## Get rid of blocks, use flexible box!

---

The "flexbox" layout mode redefines how we do layouts in CSS. You have an assignment to do *Flexbox Froggy* and *Flexbox Defense*. These teach you well how to use "flexbox". Conceptually, though, it provides a way to build rows and columns and align the elements within those rows or columns.

## The all powerful grid layout

---

The "grid" layout mode is the newest of CSS layouts. You can use it to break an element's layout space into a grid and assign child elements to sectors in the grid. You have an assignment to complete the *CSS Grid Gardento* learn how to use it.

# CSS Transitions

🕒 6 minutes

✅ Completed

## CSS Transitions

CSS transitions provide a way to control animation speed when changing CSS properties. Instead of having property changes take effect immediately, you can cause the changes in a property to take place over a period of time. For example, if you change the color of an element from white to black, usually the change is instantaneous. With CSS transitions enabled, changes occur at time intervals that follow an acceleration curve, all of which can be customized.

Animations that involve transitioning between two states are often called implicit transitions as the states in between the start and final states are implicitly defined by the browser.

CSS transitions let you decide which properties to animate (by listing them explicitly), when the animation will start (by setting a delay), how long the transition will last (by setting a duration), and how the transition will run (by defining a timing function, e.g. linearly or quick at the beginning, slow at the end).

## Defining transitions

CSS Transitions are controlled using the shorthand transition property. This is the best way to configure transitions, as it makes it easier to avoid out of sync parameters, which can be very frustrating to have to spend lots of time debugging in CSS.

You can control the individual components of the transition with the following sub-properties:

Sub-property	Definition
transition-property	Specifies the name or names of the CSS properties to which transitions should be applied. Only properties listed here are animated during transitions; changes to all other properties occur instantaneously as usual.
transition-duration	Specifies the duration over which transitions should occur. You can specify a single duration that applies to all properties during the transition, or multiple values to allow each property to transition over a different period of time.
transition-delay	Defines how long to wait between the time a property is changed and the transition actually begins.

## Examples

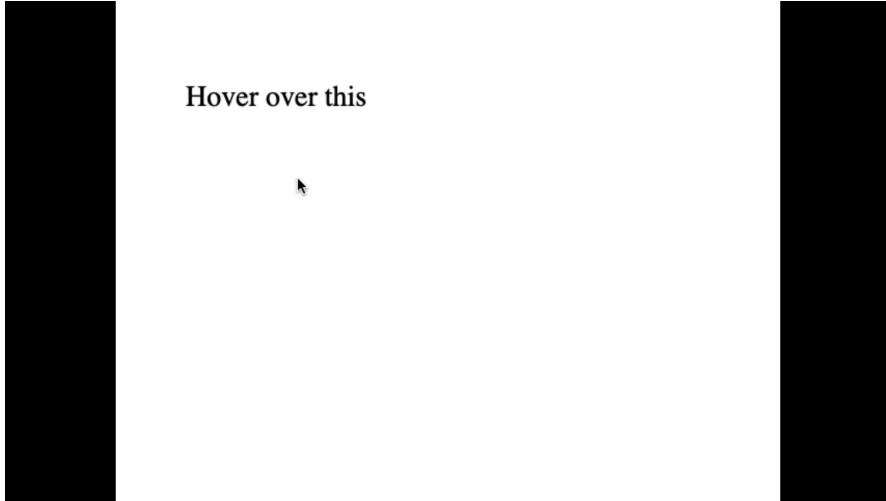
This example performs a four-second font size transition with a two-second delay between the time the user mouses over the element and the beginning of the animation effect:

```
#delay {
 font-size: 14px;
 transition-property: font-size;
 transition-duration: 4s;
 transition-delay: 2s;
}

#delay:hover {
```

```
font-size: 36px;
}
```

When the mouse hovers over it, after a delay of two seconds, a four-second transition begins which changes the font size of the text from its normal size to 36px.



In the following, any element with the "box" class will have combined transitions for: width, height, background-color, transform.

```
.box {
 border-style: solid;
 border-width: 1px;
 display: block;
 width: 100px;
 height: 100px;
 background-color: #0000FF;
 transition: width 2s, height 2s, background-color 2s, transform 2s;
}

.box:hover {
```

```
background-color: #FFCCCC;
width: 200px;
height: 200px;
transform: rotate(180deg);
}
```

When the mouse hovers over a box, it spins due to the rotate transform. Its width and height change. Its background color changes.



You can check out both of these demos at the [Transition Examples](#) on CodePen.

## What can you affect with this?

You can't apply transitions to every CSS property there is. Here is the [list of animatable CSS properties](#). If it's not in that list, then you can't animate it.

Glaringly absent from that list are any CSS properties that allow you to position elements on a Web page, properties like `left` or `bottom`. The work around is to animate the *margin* of the element.

# How to Find Help for HTML and CSS

☑ Completed

## How to Find Help for Your HTML and CSS Questions

The premier place to find answers to your questions about HTML and CSS is the [Mozilla Developer Network](#). It has really great resources on all things Web-related. Here are some shortcuts into the large library that is MDN.

- [Using HTML sections and outlines](#)
- [HTML Tables](#)
- [CSS: Cascading Style Sheets](#)
- [Use CSS to solve common problems](#)

**Warning:** *If you do a search for some HTML and get a result for `w3schools.com`, do not click on it. W3Schools has notoriously bad information, lots of ads, and some malware. Please be careful to stay away from that Web site.*

htmlreference.io is place that has a very nice, intuitive, and quick reference for how to use HTML. You can click on any tag name and get a brief example

of what you should do to make it work in a Web page. It has an advertisement on its pages, but the ad is fairly non-obtrusive.

A companion Web site is [cssreference.io](#). In the same way that the HTML site lets you click on a tag, the CSS version lets you click on a property to see how to use it. Again, minimal advertisements that you can easily ignore that allows for the continued upkeep of the Web site.

# Git Branching and Merging

🕒 8 minutes

✅ Completed

## Git Branching and Merging

The main principle of Git, once you understand it, is astonishingly simple.

Git allows you to take a snapshot of all the files in a directory. Then, if you like that snapshot, you can save it. If you don't like the snapshot, you can keep changing it until you do like it. Then, you save it.

## Working with Git

---

To start working with Git, you just need to run the “git init” command. It turns the current directory into the Git working directory and creates a repository in the ".git" (hidden) directory it creates there. You can then start working with Git. A repository is a folder whose contents are tracked by Git. It is also known as a *repo*.

A repository may have multiple files and subdirectories present within it. Usually, the files that are present within the repository contain source code.

Within each repository, there is a ".git" folder. This folder contains all the files and folders required by Git to keep track of all the changes done to the files within this repo.

If you delete the ".git" folder, Git will not identify this folder as a repo nor track its contents. You will lose all of the historical changes.

The repo present on the local computer is referred to as a local repository, and the repository located on a hosted Git platform is referred to as a remote repository.

## Initializing a Git Repository

---

In simple terms, Git converts a directory into a repository so that its contents can be tracked by it. In any directory that you want to turn into a repository, type

```
git init
```

Now, *every file and subdirectory* will be monitored for changes.

As part of this initialization process, it also creates a ".git" directory (which is hidden when you type "ls") within this repository. This contains all the files required by Git to track all the changes done to this repository.

But this is just a normal directory like other directories you have on the system. In Git terminology, you still refer to this as a repository or a local repository, to be more precise.

## Repository Status

---

At any point in time if you want to see what is being tracked by Git within a repository, you can do that by typing the command below:

```
git status
```

You will be looking at this command in more detail at some point later in the post. For now just remember, if you want to see what is being tracked within a repository by Git, you can do that using that command.

## Working with a Repository

---

Even though you have initialized the folder as a Git repository, its contents won't get tracked automatically. You need to instruct Git to take the snapshots.

In order to do that, you make use of the `git add` command. The syntax for this command is as shown below:

```
git add file [file] [file..]
```

*Note: Anything enclosed within the square brackets [] is optional. This applies to all the Git commands listed within this post. You can either specify a single file or multiple files to be tracked by Git.*

If you want Git to monitor specific files present with the repository, you can do so by specifying the individual filename of each file you would like to track.

In case you want to track files belonging to a specific file type, you can do that by specifying its file extension, as shown below. This tracks all the files ending with the .txt extension.

```
$ git add *.txt
```

If you want Git to track all the files present with the repository, the syntax is as shown below.

```
$ git add .
```

This just means "take a snapshot of the current directory (dot) and all of its contents (including all files and subdirectories) to be ready to be saved".

## Committing Staged Files

---

Now that you have a snapshot taken, you can save the snapshot to the repository. You do that by performing a "commit". You can commit these staged files by typing the command shown below.

```
git commit -m "«your message here»"
```

Whenever any change is done to a file which is being tracked by Git, you need to re-stage those files and re-commit them again. When they change, you have to take another snapshot of them and save that snapshot.

## Branching and Merging

---

If commits are snapshots, then branching is a way to take snapshots of different parallel universes. Then, when you want to combine all of them, you "merge" them back together into the main universe, the "master" branch.

Now it's time for you learn Git branching. You should do that by going to [Learn Git Branching](#) and completing the tutorial, there.





# Git Recipes

✓ Completed

## Git Cookbook

The following sections give brief “recipes” for performing common tasks using Git on the command line. This does not try to explain how each command works in detail; for that level of understanding, read the accompanying articles and watch Colt Steele's video [Learn Git in 15 Minutes](#).

### Initial setup

---

Before you run any other Git commands, you should first tell it who you are. It needs this information to assign an “author” to new commits. Run the following two commands:

```
appacademy@demo:~$ git config --global user.name "Your Name"
appacademy@demo:~$ git config --global user.email "email@example.com"
```

### Clone a repository off of the remote server

---

Most of the time, you’ll be “cloning” (copying to your local machine) a repository that is already setup on the remote server.

In the directory where you want to store your projects, run:

```
appacademy@demo:~$ git clone git@example.com:user/repo local-repo-name
```

This will create a folder in your current directory named "local-repo-name" containing the contents of the repository on the example.com server.

### Create a new repository locally

---

If you’re starting from scratch and want to add all of the files in the current directory and its subdirectories, here's that.

```
appacademy@demo:~/work/repo$ git init
appacademy@demo:~/work/repo$ git add .
appacademy@demo:~/work/repo$ git status
appacademy@demo:~/work/repo$ git commit -v -m "Initial commit."
```

### Connect your repo to a remote repo

---

Once the repository is created on the remote server, run the following within your local copy of the repository (first create a local repository as described above if you haven’t yet):

```
appacademy@demo:~/work/repo$ git remote add origin git@example.com:user/repo
```



## Normal Git workflow

---

1. Add and edit file(s)
2. Check the status of changes using `git status` or `git status -s` for a simpler version
3. Check exactly what you've changed within the files with `git diff` or `git diff --stat` for a simpler version
4. Add the changes you want to commit to the “staging area” with `git add FILE` or `git add .` if you want to stage all of the files
5. Double-check your staged changes with `git status`
6. Commit the staged changes to your local repository with `git commit -m "message"`

## Getting the latest changes from the remote server

---

This will merge any changes on the remote into your current master branch in your local working directory. Don't run this if you have uncommitted changes in your working directory.

```
appacademy@demo:~/work/repo$ git pull
```

## Branching

---

Branches are good, and with Git, they're easy to make and manage. Branches are really just lightweight movable references to commits; if you can't visualize what that means and how it affects new commits, you might have trouble using them.

To create a new branch:

```
appacademy@demo:~/work/repo$ git branch mynewbranchname
```

You can list your branches with:

```
appacademy@demo:~/work/repo$ git branch
```

You'll see that your new branch exists, but it's not yet selected. Switch to a branch with:

```
appacademy@demo:~/work/repo$ git checkout branchname
```

Switching back and forth between different branches is as easy as that. The checkout command will alter the files in your working directory to match the branch (its commit) you are switching to.

When you're done with a branch, you can delete it. This should usually only be done once that branch is merged back into another, because otherwise you're left with one or more commits that are “lost” without a name. To delete a branch:

```
appacademy@demo:~/work/repo$ git branch -d branchname
```

## Merging

---

Merge a branch `otherBranch` into the current branch with:

```
appacademy@demo:~/work/repo$ git merge otherBranch
```

If there are no conflicts, you're good. If you don't want it any more, remove the old branch with:

```
appacademy@demo:~/work/repo$ git branch -d otherBranch
```

If there are conflicts:

- See the status of the merge commit under consideration: `git status`
- Open any file with merge conflicts directly in VS Code and use its merging functionality to choose what should go in the file. Then, add and commit the files that you fixed.

# How to Find Help for Git

## How to Find Help for Your Git Questions

One of the best resources to learn branching in Git is [Learn Git Branching](#), "the most visual and interactive way to learn Git on the web; you'll be challenged with exciting levels, given step-by-step demonstrations of powerful features, and maybe even have a bit of fun along the way." I have suggested this to all of my students and new developers to learn how to interact with Git's branching mechanisms. It's truly a great resource.

Nico Riedmann wrote a comprehensive article on learning the *concepts* of Git rather than the commands. You can read his article [Learn git concepts, not commands](#). You're not using *remote* repositories, just yet. However, once you get into class, you'll be using them all the time. You can safely gloss over the remote aspects of it for the assessment; however, I would recommend that you try to learn a little bit about it to help broaden your understanding of Git.

# How to Interpret a Git Diff

🕒 15 minutes

✅ Completed

## How To Interpret A Git Diff

When you take the Git assessment, you may make a mistake. That's ok, as we've said before, because we give you four chances to pass all three Technical Challenges. But, the email that you receive will contain what's called a *diff*, which is a way that humans and computers can understand the *differences* between two files.

On each branch of the technical challenge, the grader will compare your repository's files for that branch with that of the solution's. It will make sure you have

- the correct content in each file,
- no extra files, and,
- no missing files.

It will also check that you don't have extra branches, so don't do that.

For each branch that differs, the email will tell you the name of the branch and provide the *diff*. Here's what one could look like.

```
diff --git a/site.html b/site.html
index 8dcc85..eb15129 100644
```

```
--- a/site.html
+++ b/site.html
@@ -6,11 +7,12 @@
 <meta http-equiv="X-UA-Compatible" content="ie=edge">
 <title>Art</title>
</head>
+
<body>
 <h1>Drawing and Pictures</h1>
-

+

</body>
```

The first four lines have some information about the diff, including file names of the original files and modified/new files.

On line 5, @@ indicates the start of a new *hunk*, which lasts from lines 5 through 18 of the example. Here's an explanation of all the information that you see on those lines:

- The "-6,11" means that, starting on line 6 of the solution file, there were 11 lines in the solution file
- The "+7,12" means that, starting on line 7 of *your* file, there were 12 lines in *your* work. That means that your file would have an extra line in it. The next bullet point talks about that.
- After the *hunk* marker (the stuff on the line that starts and ends with "@@"), any line that **does not start with a + or -** means that the solution file and your file are identical.
- Any line that begins with a plus (+) is a line that *your* file has that the solution does not:

- On line 9 of the example above, you see a `+` between the line that reads `</head>` and `<body>`. That's just a blank line. So, the submitted file, in this case, has an extra line between the `</head>` and `<body>` tags whereas the solution does not
- On line 15 of the example above, you see a the line `+ <img src="..."` for the Joseph Roulin drawing. That means that *your* file had that `img` tag on that line but the solution did not.
- Any line that begins with a minus (`-`) is a line that the solution file had but *your* file **did not** have. You can see on line 12 of the example above, the `- <img src="..."` line for the Joseph Roulin drawing. This tells you the solution has the `img` tag for the Joseph Roulin drawing on that line, but your file did not.

We hope this helps you understand the feedback that you get from your Git Technical Challenge. Good luck!

# Boolean Algebra

🕒 30 minutes

✅ Completed

Boolean algebra is an algebra for the manipulation of variables that can take on only two values, typically true and false, although it can be any pair of values. Because computers are built as collections of switches that are either "on" or "off", Boolean algebra is a natural way to represent digital information. In Boolean algebra, if you see the number 0, then you should interpret that as FALSE. If you see the number 1, then you should interpret that as TRUE.

## Expressions

In addition to variables, Boolean algebra also has operations that can be performed on those variables. Combining the variables and operators yields **Boolean expressions**. A **Boolean function** typically has one or more input values and yields a result based on the input values. The input values for a Boolean function are 0 and 1.

Three common Boolean operators are **AND**, **OR**, and **NOT**. To better understand these operators, you need a way to examine their behaviors. A Boolean operator can be completely described using a table that lists the inputs, all possible values for the inputs, and the resulting values of the operation for all possible combinations of these inputs. A table like this is

called a **truth table**. A truth table shows the relationship between input and output values for a specific Boolean operator or function.

Algebraists typically represent the logical operator **AND** by either a dot or no symbol at all. This is exactly like multiplication in regular-old numeric algebra, the kind you learned in school. For example, the Boolean expression  $xy$  is equivalent to the expression  $x \text{ AND } y$ . Algebraists call the expression  $xy$  the **Boolean product**. The following truth table shows the behavior of the Boolean product.

Input $x$	Input $y$	Output $xy$
0	0	0
0	1	0
1	0	0
1	1	1

The result of the expression  $xy$  is 1 only when both inputs are 1. Each row in the table represents a different Boolean expression, and all possible combinations of values for  $x$  and  $y$  are represented by the rows in the table. The first row shows where both  $x$  and  $y$  are FALSE (or 0). The last row shows where both  $x$  and  $y$  are TRUE (or 1). The middle two rows show where only one of the variables has the TRUE (or 1) value.

Algebraists usually use the plus sign to represent the Boolean operator **OR**. Therefore, the expression  $x + y$  is read  $x \text{ OR } y$ . The result of  $x + y$  is 0 only when both the input values are 0. Algebraists call the expression  $x + y$  the **Boolean sum**. Here is the truth table for the Boolean sum.

Input $x$	Input $y$	Output $x + y$
0	0	0
0	1	1
1	0	1

Input $x$	Input $y$	Output $x + y$
1	1	1

The remaining fundamental Boolean operator is the **NOT** operator. Algebraists normally write it with either an overscore or a prime (apostrophe). Therefore, both  $\overline{x}$  and  $x'$  are read **NOT  $x$** . Here is the truth table.

Input $x$	Output $x'$
0	1
1	0

Now that you understand that Boolean Algebra deals with binary values and logical operators on those variables, you can now understand how to define a Boolean function. Using the three Boolean variables  $x$ ,  $y$ , and  $z$  and the logical operators **OR**, **NOT**, and **AND**, you can define functions that represent the values of each of those variables and the resulting expression's value. Consider the Boolean function  $F(x, y, z) = x + y'z$ . To figure out the value of  $F$  for each  $x$ ,  $y$ , and  $z$  requires a truth table with three inputs and one output. You can also create intermediary columns to calculate the intermediary values  $y'$  and  $y'z$ .

$x$	$y$	$z$	$y'$	$y'z$	$F = x + y'z$
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	1	1	1
1	1	0	0	0	1

$x$	$y$	$z$	$y'$	$y'z$	$F = x + y'z$
1	1	1	0	0	1

As you can see, the last column in the truth table indicates the values of the function for all possible combinations of  $x$ ,  $y$ , and  $z$ . Note that the real truth table for the function consists of only the first three columns and the last column. The middle columns are intermediary steps to arrive at the final answer. Creating truth tables in this manner makes it easier to evaluate the function for all possible combinations of input value.

## Identities

Frequently, a Boolean expression is not in its simplest form. If you remember from numerical algebra that an expression like  $3x + 7x$  is not in its simplest form. You can reduce it to  $10x$ . You can also simplify Boolean expressions using **Boolean identities**. These identities (or laws) apply to Boolean variables as well as variable expressions.

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0 + x = x$
Null Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Inverse Law	$xx' = 0$	$x + x' = 1$
Commutative Law	$xy = yx$	$x + y = y + x$
Associative Law	$(xy)z = x(yz)$	$(x + y) + z = x + (y + z)$
Distributive Law	$x + (yz) = (x + y)(x + z)$	$x(y + z) = xy + xz$
Absorption Law	$x(x + y) = x$	$x + xy = x$
DeMorgan's Law	$(xy)' = x' + y'$	$(x + y)' = x'y'$



Identity Name	AND Form	OR Form
Double Complement Law	$x'' = x$	$x'' = x$

Due to what is known as the **duality principle**, each of the laws has an **AND** form and an **OR** form

*One of the most common errors that beginners make when working with Boolean logic is to assume the following:  $(xy)' = x'y'$ . This is wrong! Refer to DeMorgan's Law to see the correct identity. To prove it to yourself, generate the truth table for  $(xy)'$  and  $x'y'$  to see if they are equivalent.*

## Simplification example

Given the function  $F(x, y) = (xy)'(x' + y)(y' + y)$ , simplify it.

Fn	.	Expression	Reason
$F(x,y)$	=	$(xy)'(x' + y)(y' + y)$	
	=	$(xy)'(x' + y)(1)$	Inverse Law
	=	$(xy)'(x' + y)$	Identity Law
	=	$(x' + y')(x' + y)$	DeMorgan's Law
	=	$x' + y'y$	Distributive Law (AND Form)
	=	$x' + 0$	Inverse Law
	=	$x'$	Identity Law

## Complements

Boolean identities can be applied to Boolean expressions, not simply Boolean variables. The same is true for Boolean operators. The most common Boolean operator applied to more complex Boolean expressions is the **NOT** operator. With the **NOT** operator, you can get the **complement** of an expression. Quite often, it is cheaper and less complicated to implement the complement of a function rather than the function itself. If you implement the complement, you must invert the final output to yield the original function. This is accomplished with one simple **NOT** operation.

To find the complement of a Boolean function, use DeMorgan's Law. The **OR** form of the law states that  $(x + y)' = x'y'$ . You can easily extend that to three or more variables. For example, if  $F(x, y, z) = (x + y + z)'$ , then  $F'(x, y, z) = x'y'z'$ .

*The Boolean lesson to DeMorgan's Law is that to find the complement of any expression, simply replace each variable by its complement ( $x$  by  $x'$ ) and interchange **ANDs** and **ORs**.*

## Function representations

You have seen, now, that you can express Boolean functions as truth tables or different Boolean expressions. In fact, there are an infinite number of Boolean expressions that are **logically equivalent** to one another. Two expressions that can be represented by the same truth table are considered logically equivalent.

To help eliminate potential confusion, people that use logic to design solutions specify a Boolean function using a unique standardized form. The two most common standards are **sum-of-products form** and **product-of-sums** form.

The sum-of-products form requires that the expression be a collection of ANDed variables that are ORed together. The expression  $xy + yz' + xyz$  is in sum-of-products form. The expression  $xy' + x(y + z')$  is *not* in sum-of-products form. You can apply the Distributive Law to the  $x$  variable to get  $xy' + xy + xz'$  which is now in sum-of-products form.

The product-of-sums form consists of ORed variables that are ANDed together. This is more confusing and harder to use than the sum-of-products form. Most people use the sum-of-products form for that reason.

# Karnaugh Maps

🕒 30 minutes

✅ Completed

Minimizing Boolean expressions and their relationships helps reduce the total number of components that would be needed to create digital circuits, thereby lowering the cost and complexity of manufacturing. Having fewer components also allows the circuitry to run faster.

Reducing Boolean expressions can be done using Boolean identities; however, using identities can be difficult because no rules are given on how or when to use the identities, and there is no well-defined set of steps to follow. In one respect, minimizing Boolean expressions is very much like doing a proof. To alleviate this onus, there is a systematic way to reduce Boolean expressions.

## Kmaps

Karnaugh maps, or Kmaps, are a graphical way to represent Boolean functions. A map is simply a table used to enumerate the values of a given Boolean expression for different input values. The rows and columns correspond to the possible values of the function's inputs. Each cell represents the outputs of the the function for those possible inputs.

If a product term includes all of the variables exactly once, either complemented or not complemented, this product term is called a **minterm**. For example, if there are two inputs  $x$  and  $y$ , there are four minterms  $x'y'$ ,  $x'y$ ,  $xy'$ , and  $xy$ , which represent all of the possible input combinations for the function. If the input variables are  $x$ ,  $y$ , and  $z$ , then there are eight minterms.

A Kmap is a table with a cell for each minterm, which means that it has a cell for each line of the truth table for a function. Consider the function  $F(x, y) = xy$  and its truth table.

$x$	$y$	$xy$
0	0	0
0	1	0
1	0	0
1	1	1

The corresponding Kmap is

$x \backslash y \rightarrow$	0	1
0	0	0
1	0	1

Notice that the only cell in the map with a value of 1 occurs when  $x = 1$  and  $y = 1$ , the same values for which  $xy = 1$ .

Here's another example, this time for  $F(x, y) = x + y$ .

$x$	$y$	$x + y$
0	0	0
0	1	1
1	0	1

$x$	$y$	$x + y$
1	1	1

The corresponding Kmap is

$x \backslash y \rightarrow$	0	1
0	0	1
1	1	1

Three of the minterms in the previous Kmap have a value of 1, exactly the minterms for which the input to the function gives you a 1 for the output. To assign 1s in the Kmap, you simply put 1s where you find corresponding 1s in the truth table.

## Simplification in two variables

Here, again, is the Kmap for  $F(x, y) = x + y$ .

$x \backslash y \rightarrow$	0	1
0	0	1
1	1	1

To use a map to reduce a Boolean function, you simply need to group 1s in groups of 1, 2, 4, 8, and so on. Here are the rules for grouping:

- Groups can only contain 1s
- Groups cannot be diagonal
- Groups must have 1, 2, 4, 8, ... members
- Groups can only be rectangles

- Groups must be as large as possible
- Groups can overlap
- Groups can "wrap around"

That last rule is a little weird. You will see an example once you get to Kmaps for three or more variables.

To simplify using Kmaps, first create the groups as specified by the rules. After you have found all groups, examine each group and discard the variable that differs within each group. For example, the following Kmap has two groups.

$x \backslash y \rightarrow$	0	1
0	0	1X
1	1	1X

One group is the third column which has 1s in both of its cells. The other group is the third row which has 1s in both of its cells.

$x \backslash y \rightarrow$	0	1
0	0	1
1	1X	1X

Beginning with this most recent group, the cells marked "X" represent the minterms  $xy'$  and  $xy$ . That means the *group*, that is both cells, represent the statement  $xy' + xy$ . These terms differ in  $y$ , so  $y$  is discarded leaving only  $x$ . The other group represents  $x'y + xy$ . These differ in  $x$ , so  $x$  is discarded, leaving  $y$ . Thus, the two groups put together:

$$\begin{aligned}
 F(x, y) &= (xy' + xy) + (x'y + xy) \\
 &= (x) + (y) \\
 &= x + y
 \end{aligned}$$

## Simplification in three variables

You can apply KMaps to expressions of more than two variables. Here's a Kmap in three variables for  $F(x, y, z) = x'y'z + x'yz + xy'z + xyz$ .

$x,y/z \rightarrow$	00	01	11	10
0	0	1	1	0
1	0	1	1	0

The first difference that you'll notice is that the two variables  $y$  and  $z$  are grouped together in the table. The second difference is that the numbering for the columns is not sequential. Instead of labeling the columns as 00, 01, 10, 11 (the normal binary progression), they're labeled as 00, 01, 11, 10. The input values for the Kmap *must* be ordered so that each minterm differs in only one variable from each neighbor. By using the 00, 01, 11, 10 order, the corresponding minterms  $x'y'z$  and  $x'yz$  only differ in the  $y$  variable. Remember, to reduce, we need to discard the variable that is different. Therefore, we must ensure that each group of two minterms differs in only one variable.

The largest groups you can find in the two-variable example were composed of two 1s. It is possible to have groups of four or eight 1s depending on the function. For the previous Kmap, there is one square group of 1s. The fewer groups that you have, the fewer terms there will be and the simpler the expression.

Two 1s in a group allowed you to discard one variable. Four 1s in a group allows you to discard two variables. In the group of four from this example, the minterms are  $x'y'z$ ,  $x'yz$ ,  $xy'z$ , and  $xyz$ . These all have  $z$  in common, but the  $x$  and  $y$  variables differ. So, you can discard both  $x$  and  $y$  leaving you

with just  $z$ . So, you can reduce  $F(x, y, z) = x'y'z + x'yz + xy'z + xyz = z$ .

You can now learn about the "wrap around" rule. Here's a Kmap.

$x,y/z \rightarrow$	00	01	11	10
0	1	0	0	1
1	1	0	0	1

You would think that by looking at it, there are two groups. In the following table, they're marked as Xs and Ys.

$x,y/z \rightarrow$	00	01	11	10
0	1X	0	0	1Y
1	1X	0	0	1Y

Now, remember that you can group together minterms that differ by one value. If you look at the values in the outside columns, you'll see the minterms 00 and 10. Because they differ by only 1 value, you can group them together! That means there are not two groups in the Kmap, there's only one!

$x,y/z \rightarrow$	00	01	11	10
0	1X	0	0	1X
1	1X	0	0	1X

This would mean that you have the four minterms  $x'y'z$ ,  $x'yz$ ,  $xy'z$ , and  $xyz$ . Looking at those, you can see that  $z$  is the same in all of them and they differ in  $x$  and  $y$ . That means that you can discard the  $x$  and  $y$  portions. This Kmap simplifies to  $F(x, y, z) = z$ .

## Invalid inputs

There are certain conditions where a function may not be completely specified, meaning there may be some inputs that are undefined for the function. For example, consider a function with four inputs that act as bits to count, in binary, from 0 to 10 (decimal). You could use the bit combinations 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, and 1010. However, you would not use the combinations 1011, 1100, 1101, 1110, and 1111 because those are beyond the 10 values that you need. The latter inputs would be invalid which means the output of those values should not be 0 or 1. They should not be in the truth table at all.

You can use these "undefined" inputs to your advantage when simplifying Kmaps. Because they are input values that should not matter (and should never occur), you can let them have values of either 0 or 1, depending on what helps you most. The basic idea is to set these inputs in such a way that they either contribute to making a larger group or don't contribute at all.

The "undefined" inputs are typically indicated with an "X" in the appropriate cell of a Karnaugh Map.

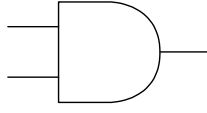
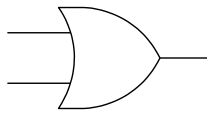
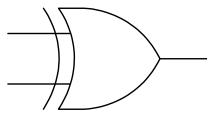
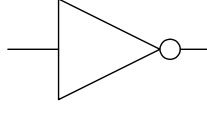
For example, the function  $F(w, x, y, z) = w'z + yz$  with undefined values for 0000, 0010, 0101, and 1100, would have the Kmap shown below. You can use the undefined values to help with minimization. If you treat the "X" values in the first row as 1s, then you can create a group of four. The other values are treated as 0s because they would not help create larger groups. (The empty cells are 0s.) You get to choose which of the Xs to replace with 1s. It's totally at your discretion.

$wx \backslash yz \rightarrow$	00	01	11	10
00	X	1	1	X
01		X	1	
11	X		1	
10			1	

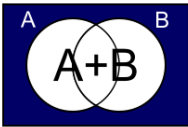
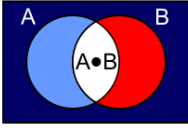
would become

$wx \backslash yz \rightarrow$	00	01	11	10
00	1	1	1	1
01			1	
11			1	
10			1	

to give two groups of four. That would yield  $F(w, x, y, z) = w'x' + yz$ . This is an acceptable function for the original Kmap because it outputs 1s for all of the original places where 1s exist.

Name	Reads as	Logic Gate	OCR Notation	Alternative Notation	Examples	Truth table	Notes															
Conjunction	AND			AND  e.g.    A AND B	Both operators in a submarine need to turn their launch keys at the same time for a missile to fire.	<table><tr><td>A</td><td>B</td><td>Output</td></tr><tr><td>0</td><td>0</td><td><b>0</b></td></tr><tr><td>0</td><td>1</td><td><b>0</b></td></tr><tr><td>1</td><td>0</td><td><b>0</b></td></tr><tr><td>1</td><td>1</td><td><b>1</b></td></tr></table>	A	B	Output	0	0	<b>0</b>	0	1	<b>0</b>	1	0	<b>0</b>	1	1	<b>1</b>	Adding a circle to the end of an AND gate turns it into a NAND gate (Not AND). It simply reverses the output from the gate.
A	B	Output																				
0	0	<b>0</b>																				
0	1	<b>0</b>																				
1	0	<b>0</b>																				
1	1	<b>1</b>																				
Disjunction	OR			OR        e.g A OR B  +        e.g A+B	The fire alarm will go off if the smoke detector senses the temperature rises to high or if it senses smoke or both.	<table><tr><td>A</td><td>B</td><td>Output</td></tr><tr><td>0</td><td>0</td><td><b>0</b></td></tr><tr><td>0</td><td>1</td><td><b>1</b></td></tr><tr><td>1</td><td>0</td><td><b>1</b></td></tr><tr><td>1</td><td>1</td><td><b>1</b></td></tr></table>	A	B	Output	0	0	<b>0</b>	0	1	<b>1</b>	1	0	<b>1</b>	1	1	<b>1</b>	Adding a circle to the end of an OR gate turns it into a NOR gate (Not OR). It simply reverses the output from the gate.
A	B	Output																				
0	0	<b>0</b>																				
0	1	<b>1</b>																				
1	0	<b>1</b>																				
1	1	<b>1</b>																				
Exclusive Disjunction	XOR			XOR        e.g A XOR B  e.g A    B	Opposing football teams will get 3 points for a win if one side scores more, but not if they draw.	<table><tr><td>A</td><td>B</td><td>Output</td></tr><tr><td>0</td><td>0</td><td><b>0</b></td></tr><tr><td>0</td><td>1</td><td><b>1</b></td></tr><tr><td>1</td><td>0</td><td><b>1</b></td></tr><tr><td>1</td><td>1</td><td><b>0</b></td></tr></table>	A	B	Output	0	0	<b>0</b>	0	1	<b>1</b>	1	0	<b>1</b>	1	1	<b>0</b>	Adding a circle to the end of an XOR gate turns it into a XNOR gate (Not XOR). It simply reverses the output from the gate.
A	B	Output																				
0	0	<b>0</b>																				
0	1	<b>1</b>																				
1	0	<b>1</b>																				
1	1	<b>0</b>																				
Negation	NOT		¬	<i>bar</i> e.g A ~            e.g ~A NOT        e.g NOT A	A microwave will stop if the door is not closed. A house alarm will go off if the door is not closed.	<table><tr><td>A</td><td>Output</td></tr><tr><td>1</td><td><b>0</b></td></tr><tr><td>0</td><td><b>1</b></td></tr></table>	A	Output	1	<b>0</b>	0	<b>1</b>										
A	Output																					
1	<b>0</b>																					
0	<b>1</b>																					
Equivalence	If and only if. Means the same as				The decimal 0.25 is the same as the fraction ¼. The temperature 25 Celsius is the same as 77 Fahrenheit.																	

Rule	Explanation	Rule	Explanation
<b>AND rules</b>	Remember that with <b>AND</b> both terms have to be 1 or <b>TRUE</b> for the result to be <b>TRUE</b>	<b>OR rules</b>	Remember that with <b>OR</b> only 1 term has to be 1 or <b>TRUE</b> for the result to be <b>TRUE</b>
1 X 0 = 0	X AND 0 is the same as 0 <i>Or to put it another way... X AND FALSE has to equal FALSE (See truth table on reverse side for proof)</i>	5 X 0 = X	X OR 0 is the same as X <i>Or to put it another way... X OR FALSE has to equal TRUE (See truth table on reverse side for proof)</i>
2 X 1 = X	X AND 1 is the same as X <i>Or to put it another way... X AND TRUE has to equal TRUE (See truth table on reverse side for proof)</i>	6 X 1 = X	X OR 1 is the same as X <i>Or to put it another way... X OR TRUE has to equal TRUE (See truth table on reverse side for proof)</i>
3 X X = X	X AND X is the same as X <i>Or to put it another way... X AND X has to equal X (See truth table on reverse side for proof)</i>	7 X X = X	X OR X is the same as X <i>Or to put it another way... X OR X has to equal X (See truth table on reverse side for proof)</i>
4 X ¬X = 0	X AND not X is the same as 0 <i>Or to put it another way... X AND NOT(X) has to equal FALSE (See truth table on reverse side for proof)</i>	8 X ¬X = 1	X OR not X is the same as 1 <i>Or to put it another way... X OR NOT(X) has to equal TRUE (See truth table on reverse side for proof)</i>

Rule	What does it mean?	In Boolean Algebra in the OCR exam	Examples in English	Notes
<b>De Morgan's Law</b>	Either logical function <b>AND</b> or <b>OR</b> may be replaced by the other, given certain changes to the equation.	$\neg (A \text{ } B) \quad (\neg A) \text{ } (\neg B)$ <b>NOT (A OR B) is the same as (NOT A) AND (NOT B)</b>  This is the same as:  $\neg (A \text{ } B) \quad (\neg A) \text{ } (\neg B)$ <b>NOT (A AND B) is the same as (NOT A) OR (NOT B)</b>	It cannot be both <i>winter</i> <b>AND</b> <i>summer</i> (at any point in time)  Is the same as...  (At any point in time) It is <b>NOT</b> <i>winter</i> <b>OR</b> it is <b>NOT</b> <i>summer</i>	 
<b>Distribution</b>	This law allows for the multiplying or factoring out of an expression.	This is the <b>OR</b> Distributive law: $A \text{ } (B \text{ } C) \quad (A \text{ } B) \text{ } (A \text{ } C)$ <b>A AND (B OR C) is the same as (A AND B) OR (A AND C)</b>  This is the <b>AND</b> Distributive law: $A \text{ } (B \text{ } C) \quad (A \text{ } B) \text{ } (A \text{ } C)$ <b>A OR (B AND C) is the same as (A OR B) AND (A OR C)</b>	You can choose 1 main course and either a start or a desert.  Is the same as...  You can choose 1 main and 1 starter or you can choose 1 main and 1 desert	
<b>Association</b>	This law allows for the removal of brackets from an expression and the regrouping of the variables.	This is the <b>OR</b> Association Law: $A \text{ } (B \text{ } C) \quad (A \text{ } B) \text{ } C \quad A \text{ } B \text{ } C$ <b>A OR (B OR C) is the same as (A OR B) OR C is the same as A OR B OR C</b>  This is the <b>AND</b> Association Law: $A \text{ } (B \text{ } C) \quad (A \text{ } B) \text{ } C \quad A \text{ } B \text{ } C$ <b>A AND (B AND C) is the same as (A AND B) AND C is the same as A AND B AND C</b>	“Craig and his friends James & Tom are coming to the party”  Is the same as..  “James & Tom and their friend Craig are coming to the party”  Is the same as...  “Craig, James and Tom are coming to the party”	
<b>Commutation</b>	The order of application of two separate terms is not important.	$A \text{ } B \quad B \text{ } A$ The order in which two variables are <b>AND</b> 'ed makes no difference  $A \text{ } B \quad B \text{ } A$ The order in which two variables are <b>OR</b> 'ed makes no difference	Tom and Jane are going shopping.  Is the same as...  Jane and Tom are going shopping	
<b>Double negation</b>	NOT NOT A (double negative) = "A"	$\neg(\neg A) = A$	“It's not as if I don't like you” clearly means “I do like you”!	
<b>Absorption</b>	Where the rule applies the second term inside the bracket can always be eliminated and “absorbed” by the term outside the bracket.	$X \text{ } (X \text{ } Y) \quad X$ <b>X OR (X AND Y) is the same as X</b>  $X \text{ } (X \text{ } Y) \quad X$ <b>X AND (X OR Y) is the same as X</b>	If it will rain, then I will wear my coat.  Therefore, if it will rain then it will rain and I will wear my coat.	To be able to apply the Absorption rule: 1. The operators inside and outside the brackets must be different 2. The term outside the brackets must also be inside the brackets

In addition to the 5 laws / rules above which are listed in the OCR specification and the Absorption rule there are also 8 general identities or “rules” which you really should know which will help you gratefully when it comes to simplifying Boolean Expressions.



# How to Find Help for Boolean Logic and Digital Circuits

- [Logic Gates](#)
- [Combinational Circuits](#)
- [Sequential Circuits](#)

## How to Find Help for Your Boolean Logic and Digital Circuits Questions

Bill MacKenty teaches computer science at the American School of Warsaw. He's a lifelong (self-proclaimed) "geek" who likes sharing his passion, energy, and expertise about programming and computer science. He has a computer science wiki that you can access. You can find more information on [Boolean operators](#).

The Public Broadcasting Service has an online series called "CrashCourse". It has a nice video on Boolean logic and logic gates named [Boolean Logic & Logic Gates: Crash Course Computer Science #3](#). Give it a watch for a nice explanation of this information.

The incomparable Brady Haran of Numberphile fame has as sister video series named Computerphile. It has a nice video named [AND OR NOT - Logic Gates Explained - Computerphile](#) that you can watch.

Tutorials Point has a nice reference for most of what you've learned. Check them out at

- [Boolean Algebra](#)
- [K-Map Method](#)