

## **Module 2: Git and Bash Fundamentals**

**Module 2 Objective:** Learning to navigate and create project structure through the terminal, and to save and share code with git and github.

### **WHAT IS BASH**

You typically interact with your computer through a **Graphical User Interface or GUI**.

This happens when you create a new folder, save a file, open a file, search for things on your hard drive, etc.

Think of this as speaking with your computer through an interpreter. GUI's basically visualize a set of sent commands, to be executed by your computer with the files on your hard drive.

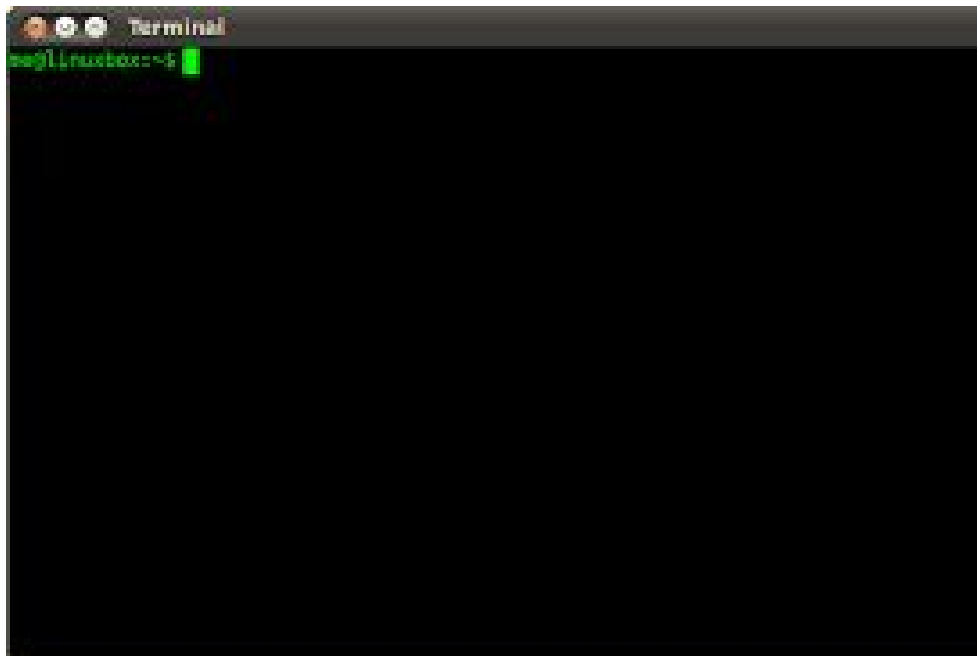
This can include:

- Creating files and directories (folders)
- Opening a directory to view its content
- Changing permissions (making a file "admin only" for example)
- Changing file names
- Copying or moving files
- Renaming files and directories
- Deleting things
- Basically everything you can do on a computer outside of the browser and installed programs

However, as a programmer you need to lose this crutch. The interpreter (GUI) is for those who do not want to speak to the computer directly and this is not you. Programmers talk to computers directly through the Terminal.

Ever seen programmers typing obscure commands into a black screen? Yeah? That black screen is the terminal. Working with the terminal seems terrifying for people but it actually is much more efficient, and over time we almost guarantee you will prefer it to regular GUIs for creating file structures for your projects.

**Say hello to the terminal:**



### **How to access your terminal:**

For Mac/Linux users you are in luck! Generally speaking, there are two types of terminals; the BASH terminal which comes standard on Macs, Linux machines, and newer versions of Windows. This handles Unix commands and is pretty standard for programmers.

For Windows users, you have the Windows CMD prompt, which has its own set of parallel commands. **However**, Windows supports an application called Powershell which basically runs all the unix BASH commands the terminal does and works largely the same way.

Newer Windows versions (10+) have their own BASH you can get from the app store, but to be safe we will go over how to install Powershell since it works on all recent windows machines.

### [Powershell vs. BASH](#)

### **Opening Powershell:**

---

**Windows 10**

**Click left lower corner Windows icon, start typing PowerShell**

---

**Windows 8.1, 8.0**

**On the start screen, start typing PowerShell.**

---

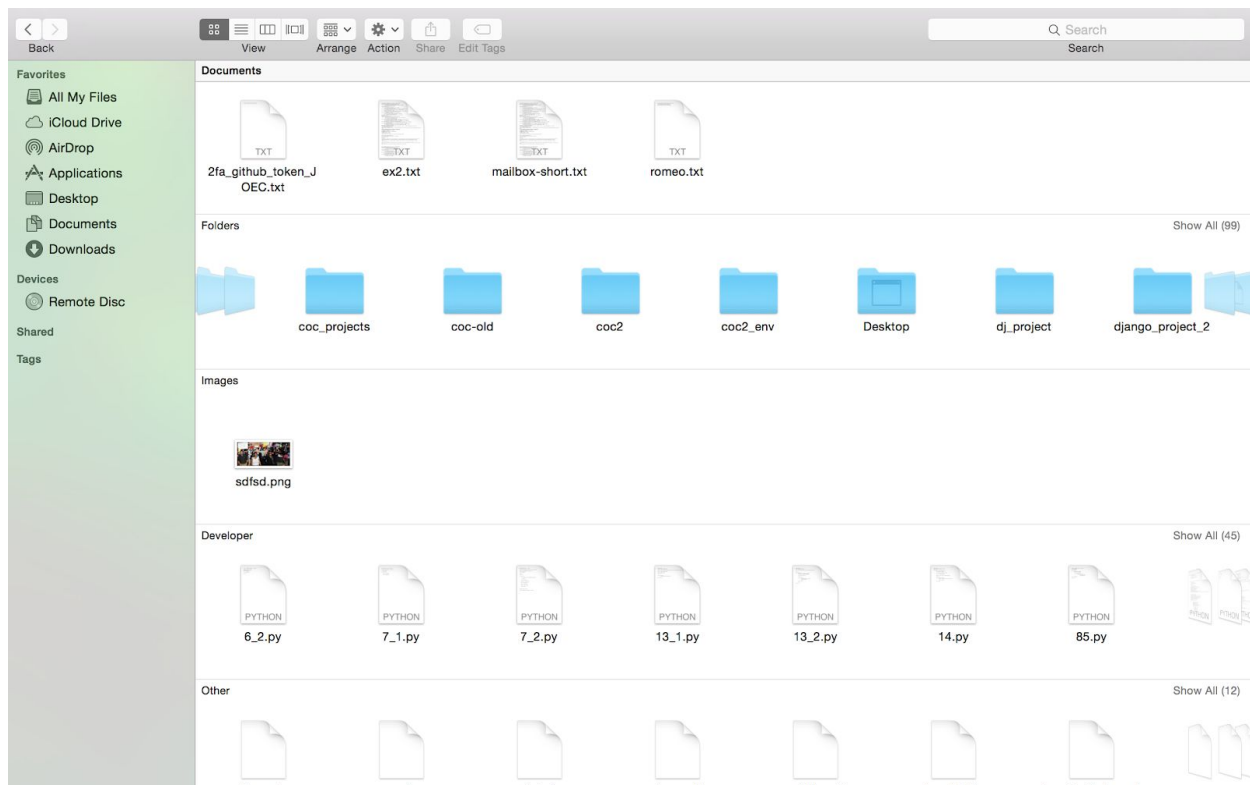
**If on desktop, click left lower corner Windows icon, start typing PowerShell**

## Opening Terminal on MacOS:

Go to the search icon in the top right corner of the nav bar; type in “terminal”

**When you open your terminal whether bash or powershell it opens to a location on your computer.**

In bash on Linux and MacOS the home folder is the folder associated with your username, this contains your downloads and documents folder.



```

Last login: Mon Mar 2 12:55:57 on console
-bash: o: command not found
-bash: /Users/joeknows718/virtualenv-auto-activate.sh: No such file or directory
/Library/Frameworks/Python.framework/Versions/2.7/Resources/Python.app/Contents/MacOS/Python: No module named virtualenvwrapper
virtualenvwrapper.sh: There was a problem running the initialization hooks.

If Python could not import the module virtualenvwrapper.hook_loader,
check that virtualenvwrapper has been installed for
VIRTUALENVWRAPPER_PYTHON=/Library/Frameworks/Python.framework/Versions/2.7/bin/python and that PATH is
set properly.
Joes-MBP:~ joeknows718$ ls
13.1.py                ether
13.2.py                ex.py
14.py                  ex2.txt
2                       exJson.py
2Fa_github_token_JOEC.txt  exercism
6_2.py                 ex11.py
718178.py              exsml.py
718_Digital            fakeblog
7.1.py                 file.py
7.2.py                 flaskong
85.py                  flask_w_class
Adobe Illustrator CS6 16.0.0 Final (Eng Jpn) Mac Os X [ChingLiu]  for_index.py
Adobe Photoshop CS6 13.0 Final (English Japanese) Mac Os X [ChingLiu] geocode.py
Adv_JS                 greatest.py
Advanced_JS            hello-next
Agres.db               html_ex
Android_SDK            hw_16.db
AndroidStudioProjects hw_16.db-journal
Applications           import urllib.py
Applications (Parallels) jsonn.py
CoC_Codebase           leaderboard
Desktop               list.py
Documents             mailbox-short.txt
Downloads              match3.py
ENV                   microblog
Envs                   music.db
Express-Projects       nameless-fjord-82483
IACC_Landing           newenv
Into-Lite-Workshops   nextjs
Library               node-class-projects
Movies                node_modules
Music                 num_list.py
Pictures              obj.py
Public                opentkh.py
React-Fundamentals     other
RestaurantWebApplication package-lock.json
SYEP                  package.json
TKH-site              philly_db
TKHV2_Offical         practice_dir
TLW                   project_house
TagsAreKnown          projects
Team_Project          prosecutor_db
Tut2718               py_class
Vagrantfile           pyethapp
VirtualBox VMs        pyethereum

```

These screens represent the same thing. One is through the GUI and one is through the terminal.

If you are ever confused about where your terminal opens up, you can print your home folder like so:

```

#print home folder
$ echo $HOME
#print current folder
$ echo $PWD

```

**Example 1: Open your terminal, print your home, print your working directory**

## Navigation through the terminal:

Just like your GUI you can navigate folder to folder in bash using the command “cd” (Change Directory).

```

$ cd Desktop
#changes directory to desktop
$ cd Desktop/Project1

```

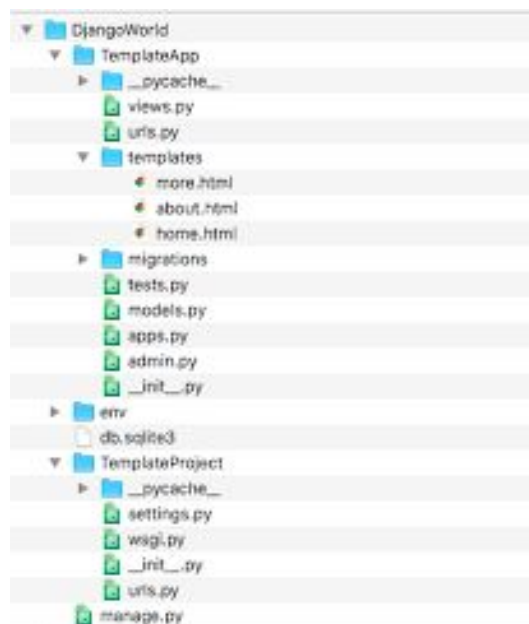
```
#skips two folders over
#if you did this you would be in homefolder/Desktop/Project1
#If you wanted to skip back to Desktop you could go back with the following
command:
$ cd ..
#To jump back to home, you can skip two folders back:
$ cd ../../
#this would land you back at home
#also note you can autofill folder names by hitting the tab once you start
#writing the name of the directory
$ cd .
#takes you back to home
#so $ cd De + TAB would autofill Desktop
```

**Since we know where we are, let's make some stuff:**

The main way you will be using the terminal is going to be creating project folders and file structures.

One of the keystones to being a developer is following best practices in setting up your projects.

### Django project folder structure



You will be setting these up through the terminal, doing so from the GUI is not a best practice and is not an acceptable move professionally.

So let's talk about setting your project up...

**Here are a few key commands:**

```
#creating a new folder with the make directory command:  
$ mkdir FOLDERNAME  
#this creates a new directory(folder) called FOLDERNAME in the directory  
(folder) you are #currently working in. So doing this in your homefolder  
will result in homefolder/FOLDERNAME  
#being created
```

Next we want to make some files:

```
#We make files by using the touch keyword  
$ touch filename  
#this creates a new file in whicher your working directory is
```

You can also list out the contents of whatever directory you are in:

```
$ls  
#for Bash  
$ dir  
#for powershell
```

Moving folders and files also might be necessary:

```
$mv /homefolder/filename /homefolder/Documents  
#the first argument taken is the location of the file to move, the second  
argument is the files #destination  
#this command moves the filename file and moves it from  
#the home folder to the Documents folder  
$mv /homefolder/filename2 /homefolder/filename3  
#this command simply renames the file because you are moving the file  
contents to a new file with a new name at the same directory location.
```

These are some of the most commonly used commands for setting up your developer environments and files structures, but there are more.

```
#Here are some common commands you might want to use:
```

```
$ clear
#this clears out your terminal window
$ cp filelocation/filename newlocation/newname
#above commands copies files and directories and places it in a new
location
#Lastly is the alias command that will allow you to save common commands as
whatever you #want
$alias c = "clear"
#now you can call the clear command by typing c into the terminal
```

**Exercise 2: Having learned the basics of using the terminal to talk to your machine, we are going to set up our first file structure. The folder setup we will be creating will be the first step in submitting your modules through Git which we will be covering next.**

Go to your home folder and create a new directory called TKH\_Modules. Change directories into this folder and create a directory for each of your 7 modules with the following names:

Module\_1\_What\_is\_Code  
Module\_2\_Git\_Bash  
Module\_3\_HTML  
Module\_4\_CSS  
Module\_5\_Programming1  
Module\_6\_Programming2  
Module\_7\_Problem\_Solving

Now move the .txt file with your responses to module 1 into the Module\_1\_What\_is\_Code folder.

**Your end result should look like this:**

- TKH\_Modules
  - Module\_1\_What\_is\_Code
    - What\_is\_code\_reflection.txt
  - Module\_2\_Git\_Bash
  - Module\_3\_HTML
  - Module\_4\_CSS
  - Module\_5\_Programming1
  - Module\_6\_Programming2
  - Module\_7\_Problem\_Solving

## **GIT FUNDAMENTALS**

## What is Git?

**Git is the most commonly used version control system for developers period, and GitHub, a public place to host Git Repos is basically the biggest single online asset a budding developer needs to get noticed by employers.**

### **What is a version control system:**

A version control system is a tool for managing changes to your codebase (the folders and files your code is written in) in a way that allows for changes to your code to be tracked over time, and makes code being worked on by more than one person available in a central place.

Basically it's Google Docs for code: it allows multiple people to work on the same files, and for changes to be tracked, enabling developers to compare each other's changes and revert projects back to a previous version if the current version is broken.

**Git also uses a branching system. Branches are ways to maintain different versions of your codebase at the same time.**

Let's say we have a codebase for our app creatively named APP. Usually a project would have at least 3 working branches.

The default branch is called MASTER and is the first branch created when a new git repo is created. Usually Master is where your finished and tested changes would go, to be deployed to the web and ultimately your users.

Next we have staging. Staging tests code's functionality and are integrated and tested in a full system.

Lastly we have feature branches, where new features in development are saved and tested before being ready to be integrated into the larger project.

**Example:** A team on an e-commerce site is looking to build out a new shopping cart experience for customers.

The team developing this new feature will most likely be working off a Git branch called **cart**. Here the team will build and test code running only on their individual computers, until the cart is ready to be tested on a live staging site.

This transfer would happen by **MERGING** branches, combining the cart branch into the staging branch, with the team working through any conflicts that arise.



Once the new cart system is working on the developers' local machines, the merge would allow them to take this new feature and test if it works on the live site. However, often when pushing from a local machine to one hosted on the web, things break.

**This is why we use a staging branch which is connected to a staging server.** A staging server is a web host that puts up a private (sometimes password protected and sometimes not) clone of the live site, which while on the web is really only accessed internally. Here we can test new things in a live environment before putting them out for the world to see.

Once things have been tested and are shown to work on staging, we would wait till the app has some downtime (say 4:30am on a Sunday) and merge our staging and master branches, updating the master and pushing the changes to the actual live site.

**This explanation shows the why, but let's talk a bit about the how.**

First things first.... a note

### **GIT is not GitHub**

This is a rookie mistake but please note these are not the same thing. Git is the actual version control tool to create repos or repositories which are basically just folders that use git to track changes to the files and folders within.

If we were to activate git in your TKH\_Modules directory for example git would track any changes made to any files in that folder or the folders contained in that folder.

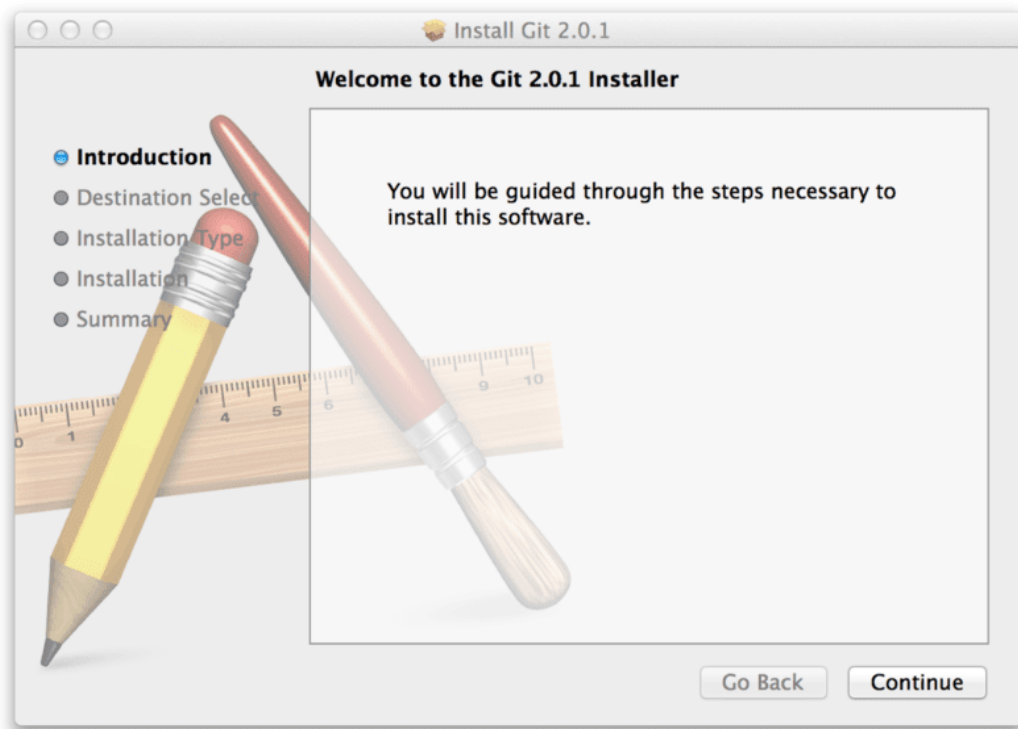
GitHub is just a platform used to save and store your code, and track its versions in a more visual way. Also Github is used to collaborate with others by providing users with a code base that is accessible by all team members. **Think working on a google doc. vs working on a word doc.**

Changes made on any team members' computer can be pushed up to a central location hosted on Github.

**It is important to note that Github is not the only place to host Git Repos. Alternatives like BitBucket are also viable. However, Github is the most popular Git Hosting/Sharing Platform**

### **Installing GIT:**

Installing on MacOS: Just type git into the terminal, if you have git, it will print out all the git command options, if not it will just prompt you to download.



### Installing on Windows:

Just go to <https://git-scm.com/download/win> and the download will start automatically.

Ex 3 Download Git if you do not have it.

Ex 4 Signing up for github: <https://github.com>

Now here are some of the basic commands for initializing a git repo (taking a folder and allowing git to track it:

```
$git init /directory_name
#for specific directory
$git init
#initializes hit in your current working directory
```

Now there are three ways to connect a repo to github for online tracking

- 1) You create an empty repo on github, and then connect it to an existing git repo on your computer.
- 2) You create an empty repo on github, and clone that repo onto your computer

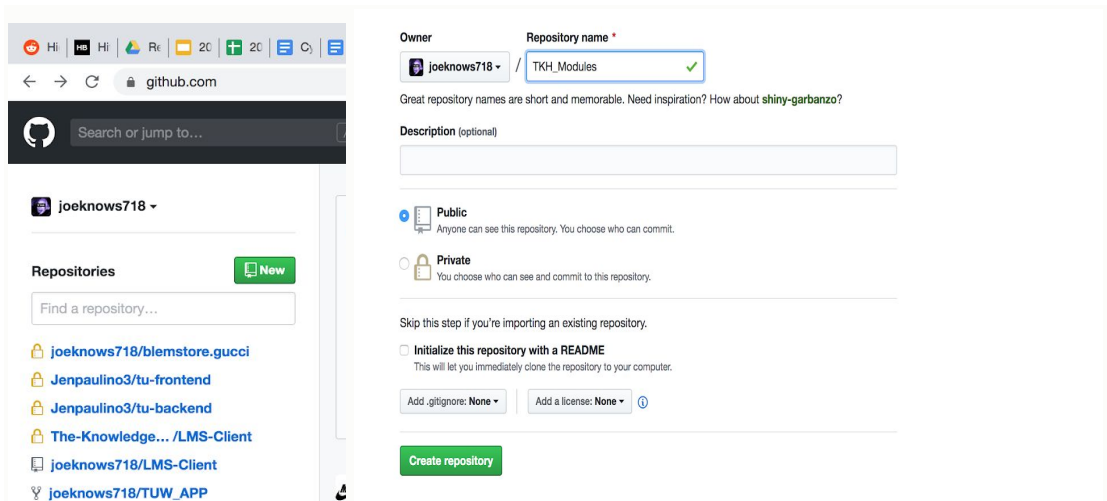
You create an empty repo on github, and then connect it to an existing git repo on your

## computer:

First step is you go into your terminal and navigate into the home directory you want to initialize your git repo in.

```
$ git init
```

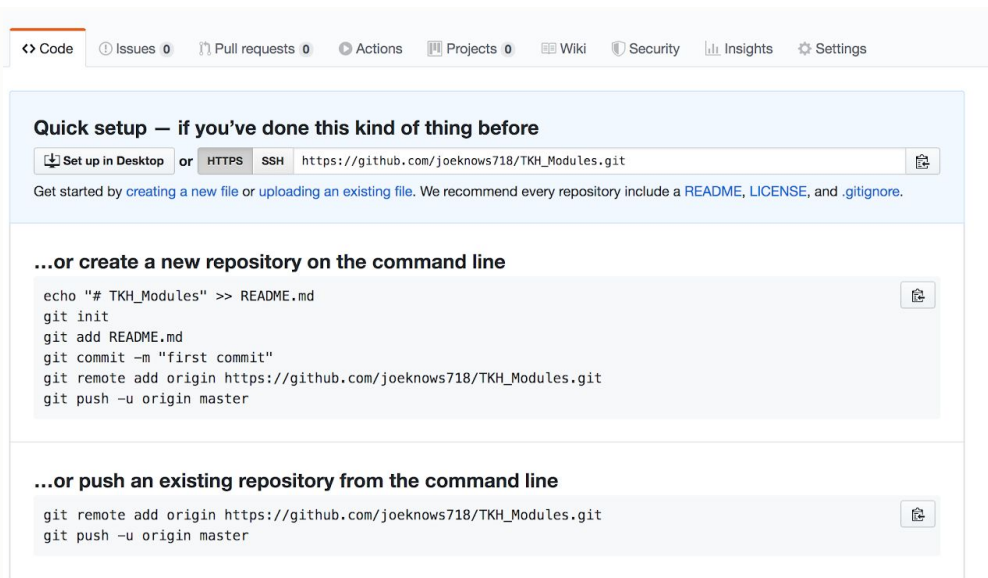
Next you need to create a repo in Github



The screenshot shows the GitHub 'Create repository' interface. On the left, there's a sidebar with the user's profile 'joeknows718' and a list of existing repositories. The main area is titled 'Create repository' and contains the following fields and options:

- Owner:** joeknows718
- Repository name:** TKH\_Modules (with a green checkmark)
- Description (optional):** An empty text box.
- Visibility:** ☒ Public (Anyone can see this repository. You choose who can commit.) and ☐ Private (You choose who can see and commit to this repository.)
- Initialization:** ☒ Initialize this repository with a README (This will let you immediately clone the repository to your computer.)
- Buttons:** 'Add .gitignore: None' and 'Add a license: None'.
- Create repository:** A green button at the bottom.

Once it is created it will give you the remote location link for your repo



The screenshot shows the GitHub repository page for 'TKH\_Modules'. The top navigation bar includes links for Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. The main content area is titled 'Quick setup — if you've done this kind of thing before' and contains the following information:

- Set up in Desktop:** A button to set up the repository on a desktop.
- HTTPS:** A link to the repository: `https://github.com/joeknows718/TKH_Modules.git`.
- Get started by:** A note that recommends including a README, LICENSE, and .gitignore.
- ...or create a new repository on the command line:** A section with a copy icon and the following commands:

```
echo "# TKH_Modules" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/joeknows718/TKH_Modules.git
git push -u origin master
```
- ...or push an existing repository from the command line:** A section with a copy icon and the following commands:

```
git remote add origin https://github.com/joeknows718/TKH_Modules.git
git push -u origin master
```

Now we have already created our file structure so we want to connect our TKH\_Modules folder to our new TKH\_Modules Repo:

```
$cd /THK_Modules
$git init
#we initialize our git repo
$git add .
#the add command takes everything we have in our directory and adds them to be
#tracked by git
$git commit -m "first commit"
#git only pushes changes as 'commits' along with a message describing what the
#changes are. You can not push to git without a message
$git remote add origin <YOUR GITHUB REPO URL>
#this command connects your local git repo to the empty github repo you created
earlier
$git remote -v
#this commands will print out the remote connection for your repo, if the
#previous step worked the address of your github repo should show up as origin
#master
$ git push origin master
#pushes all files and folders to github
```

Now we have successfully pushed to our repo.

Congrats you're using Git!

## Git Branching

As mentioned earlier, not only do we save things to our git repos but we also create branches because nothing should ever be sent to the main branch unless it always works. What a branch does is essentially create two separate repos for the same code base. Think of it as having two saved games, one saved where you test new stuff, and one that's your official profile.

In our module submissions we want a different branch for each homework submission so we can track your submissions from branch to branch. We also want your Master branch to be updated.

Your repo should have the following branches at the end:

Master

Module 1: Empty set of folders with the txt responses from module 1 in the module 1 folder

Module 2: Folder structure with your module two submission

Module 3: Folder structure with your module three submission

So on to module 7

Your goal will be at the end of each module to create, check out and submit your work on a specific branch and merge that branch back into the master, updating your master.

First, let's create a new branch for module 1 submissions, push to that branch and then merge that branch into master.

```
$git checkout -b Module1
#this creates and checks us into a new branch called Module 1, we now have
two branches
# Master and Module1, and we are currently checked into Module1 meaning
changes we make #will be saved in module 1 unless explicitly told not to
be.
$git branch
#this will list out all our active branches
$git add .
$git commit -m "pushing to module 1 branch"
$git push origin Module1
#pushes your files in their current state to the Module 1 Branch
```

Now we want to merge Module1 into our Master branch

```
$git checkout Master
#first we checkout our Master branch so that we move from working in our
Module1 branch back into master
$git merge Module1
#because we checkout Master, this command takes Module1 and merges it into
the branch we are currently checked into, in this case master.
```

The last thing we should touch on for now is README files. These are basically files located in the home directory of a repo (the one you used git init in) that allows Github to show some title and description info for your project. Below are some github README best practices and how to's:

<https://guides.github.com/features/wikis/>

Now that we have a strong idea about git fundamentals here is a resource for more git commands:

<https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet>

**Ex2:**

- 1) **We are going to make two files 1 is a README File which will add text and titles to our github repo.**

- 2) We will create a module2.txt file in the Module 2 folder in it you will write in your own words the answer to the following 3 questions:
  - a) What is Git and how is it different than github
  - b) Why use the terminal
  - c) Explain 3 benefits version control in your own words
- 3) Create a new branch Module2, check into it and push the new files.
- 4) Merge Module2 into Master.