Harvard’s word to enlighten yourself

CS50’s Web Programming with Python and JavaScript

GIT

Git is a command line tool that will help us with **version control** in several different ways: Allowing us to keep track of changes we make to our code by saving snapshots of our code at a given point in time. Allowing us to easily synchronize code between different people working on the same project by allowing multiple people **to pull** information from and **push information** to a **repository stored on the web**.

Allowing us to make changes to and test out code on a **different branch** without altering **our main code base**, and then **mergin**g the two together. Allowing **us to revert** back to earlier versions of our code if we realize we’ve made **a mistake**.

A Git repository is a file location where store all of the files related to a given project. These can either be **remote (stored online**) or **local (stored on my pc).**

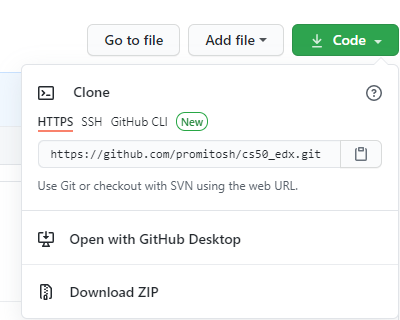
* Keep track of changes to code
* Synchronizes code between different people
* Test changed code without losing the original
* Revert back to old version of code

**[Git](https://cs50.harvard.edu/web/notes/1/" \l "more-on-git)** hub[, GitHub](https://www.github.com/) is a website that allows us to store Git repositories remotely on the web. after a GitHub account set up.

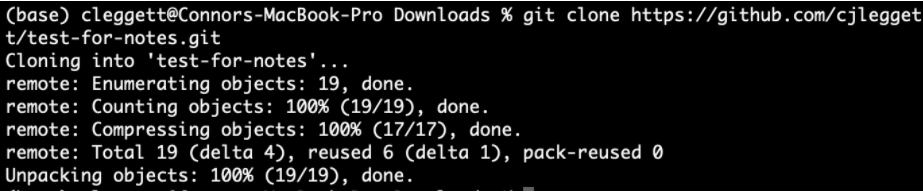
* 1. Click the **+** in the top-right corner, and then click “**New repository**”
  2. **Create a repository name** that describes your project
  3. (Optional) Provide a description for your repository
  4. Choose whether the repository should be **public** (visible to anyone on the web) or **private** (visible just to you and others you specifically grant access)
  5. (Optional) Decide whether you want to add a **README,** which is a file describing your new repository.

Once we have a repository, we’ll probably want ***to add some files*** to it. In order to do this, we’ll take our newly created **remote repository** and create a **copy, or clone**, of it as a local  repository on our computer.

* Make sure you have **git installed** on your computer **by typing git into your terminal**.
* **Click the green “Clone** *or Download” button* on your repository’s page, and copy the url that pops down.



* If you didn’t create a README, this link will appear near the top of the page in the “Quick Setup” section.
* In terminal, (or on GIT Base)run **git clone <repository url>.** This will download the repository to your computer. If you didn’t create a README, you will get the warning: You appear to have cloned into an empty repository. This is normal, and there’s no need to worry about it.



1. Run **ls** on Git terminal which is a command that **lists** all files and folders in your current directory. To see the name of the repository which just cloned.
2. Run**cd** <repository name> to **change directory** into that folder. e.g. cd C:XAMPP\htdocs\php\php
3. Run**touch** <new file name> to create a new file in that folder. to make edits to that file. **Alternatively**, open the folder in your text **editor** and manually add new files.
4. Run **git add <new file name>** to track that specific file, or git add . to track all files within that directory.
5. **Commits**, after some changes to a file, we can commit those changes, taking a snapshot of the current state of our code. To do this **git commit -m** "some message" where the message describes the changes just made.
6. After this, we can run **git status** to see how our code compares to the code on the **remote repository**
7. When ready to publish our local commits to Github, run **git push**.
8. If only changed existing files and not created new ones, instead **of using git add**, then git commit..., we can **condense** this into one command**: git commit -am "some message**". This command will commit all the changes that you made.
9. Sometimes, the remote repository on GitHub will be more up to date than the **local version**. In this case, first **commit any** changes, and then **run git pull** to pull any remote changes to repository.

**Merge Conflicts**

One problem that can emerge when working with Git, especially when working on collaborating with other people, is something called a **merge conflict**. **A merge conflict** occurs when two people attempt to change **a** **file i**n a ways that **conflict** with each other.

* This will typically occur when you either git push or git pull. When this happens Git will automatically change the file into a format that clearly outlines what the conflict is. Here’s an example where the same line was added in two different ways:

a = 1

<<<<< HEAD

b = 2

===== merge conflict

b = 3

>>>>> 56782736387980937883

c = 3

d = 4

e = 5

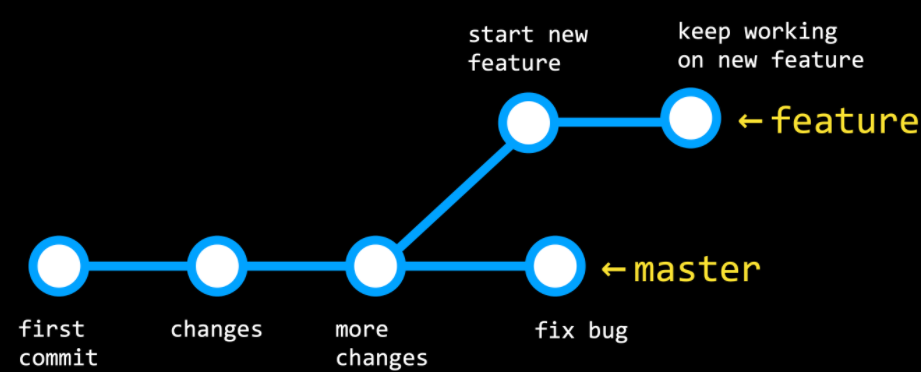
* In the above example added the line b = 2 and another person wrote b = 3, and now we must choose one of those to keep. The **long number** is a ***hash***  that represents **the commit** that is conflicting with edits. Many text editors will also provide highlighting and simple options such as “**accept current**” or “**accept incoming**” that save the time of deleting the added lines above.
* git command **is git log**, which gives a history of all of your commits on that repository.

Potentially even more helpful, if you realize that you’ve made a mistake, you **can revert back** to a previous commit using the command **git reset** in **one of two ways**:

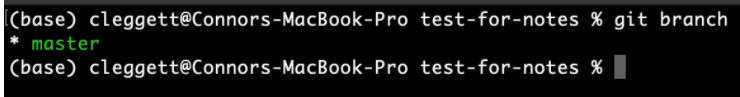
* git reset --hard <commit> reverts your code to exactly how it was after the specified commit. To specify the commit, **use the commit hash associated with a commit** which can be found using git log as shown above.
* git reset --hard origin/master reverts code to the version currently stored online on Github.

## [Branching](https://cs50.harvard.edu/web/2020/notes/1/#branching)

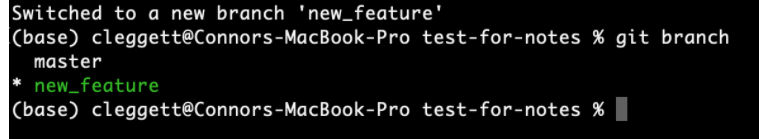
After you’ve been working on a project for some time, you may decide that you want to add an **additional feature**. At the moment, we may **just commit changes** to this new feature as shown in the graphic below but this could become **problematic** if we then discover **a bug in our original code**, and **want to revert back without changing the new feature**. This is where **branching** can become useful.



Branching is a method of moving into a new direction when creating a new feature, and only combining this new feature with the main part of your code, or the main branch, **once you’re finished**. This **workflow** will look more like the above graphic: The branch is currently looking at is determined by the HEAD, which points to one of the two branches. By default, the **HEAD is pointed at the master branch**, but we can check out other branches as well. Now, let’s get into how we actually implement branching in our git repositories:

**Run git branch** to see which branch you’re currently working on, which will have an **asterisk t**o the left of its name. 

To make a new branch, we’ll run **git checkout -b <new branch name>**



Switch between branches using the command **git checkout <branch name>** and **commit** any changes to each branch. When we’re ready **to merge** our two branches together, we’ll check out the branch we **wish to keep** (**almost always the master branch**) and then run the command **git merge <other branch name>.** This will be treated similarly **to a push or pull**, and merge conflicts may appear.

**GitHub Features More,** here are some useful features specific to GitHub that can help:-

**Forking:**

As a GitHub user, **you have the ability to fork** any repository that you have access to, which creates a **copy of the repository** that you are the owner of. We do this by clicking the “Fork” button in the top-right.

আমরা উপরের ডানদিকে "কাঁটাচামচ" বোতামটি ক্লিক করে এটি করি।

**Pull Requests**: Once you’ve forked a repository and made some changes to your version, you may want to request that those changes be added to the main version of the repository. For example, if you wanted to add a new feature to Bootstrap, you could fork the repository, make some changes, and then submit a pull request. This pull request could then be evaluated and possibly accepted by the people who run the Bootsrap repository. This process of people making a few edits and then requesting that they be merged into a main repository is vital for what is known as open source software, or software that created by contributions from a number of developers.

GitHub Pages: GitHub Pages is a simple way to publish a static site to the web. (We’ll learn later about static vs dynamic sites.) In order to do this:

Create a new GitHub repository.

Clone the repository and make changes locally, making sure to include an index.html file which will be the landing page for your website.

Push those changes to GitHub.

Navigate to the Settings page of your repository, scroll down to GitHub Pages, and choose the master branch in the dropdown menu.

Scroll back down to the GitHub Pages part of the settings page, and after a few minutes, you should see a notification that “Your site is published at: …” including a URL where you can find your site!

That’s all for this lecture! Next time, we’ll be looking at Python!

* git fetch : download all of the latest commits from a remote ‘Branching’ is a feature of Git that allows a project to move in multiple different directions simultaneously. There is one master branch that is always usable, but any number of new branches can be created to develop new features. Once ready, these branches can then be merged back into master.
* When working in a Git repository, HEAD refers to the current branch being worked on. When a different branch is ‘checked out’, the HEAD changes to indicate the new working branch.
* When merging a branch back into master, there is the possibility for merge conflicts to arise. These can be resolved in the same way discussed in Lecture 0.
* Some Git commands related to branching:
  + git branch : list all the branches currently in a repository
  + git branch <name> : create a new branch called name
  + git checkout <name> : switch current working branch to name
  + git merge <name> : merge branch name into current working branch (normally master)
* Any version of a repository that is not stored locally on a device is called a ‘remote’. ‘Origin’ is used to refer to the remote from which the local repository was originally downloaded from.
* Some Git commands related to remotes:
  + to a local device
  + git merge origin/master : merge origin/master, which is the remote version of a repository normally downloaded with git fetch, into the local, preexesiting master branch
    - Note that git pull is equivalent to running git fetch and then git merge origin/master
* A ‘fork’ of a repository is an entirely separate repository which is copy of the original repository. A forked repository can be managed and modified like any other, all without affecting the original copy.
* Open source projects are often developed using forks. There will be one central version of the software which contributors will fork and improve on, and when they want these changes to be merged into the central repository, they submit a ‘pull request’.
* A pull request can be made to merge a branch of a repository with another branch of the same repository or even a different repository. Pull requests are a good way to get feedback on changes from collaborators on the same project.
* Note that forks and pull requests are both GitHub specific features.

[**More on HTML**](https://cs50.harvard.edu/web/notes/1/#more-on-html)

* More useful HTML tags:
  + <a href="path/to/hello.html">Click here!</a> : link to hello.html, some URL, or some other content marked by id by passing #id to href
  + <input type="radio"> Option 1 : radio-button option for a form, where only 1 out of all the options may be selected ``` html
* There are lots of new useful tags with HTML5, but not all browsers, especially older browsers, will support these new features. Nonetheless, these new features can be used with increasing confidence that they will be rendered appropriately for a significant portion of users.

[**More on CSS**](https://cs50.harvard.edu/web/notes/1/#more-on-css)

* CSS selectors are used to select different parts of a website to style in particular ways.
* Some common CSS selectors:
* Select h1 and h2
* h1**,** h2 {
* **color**: red;
* }
* Select all li that are descendants of ol (not necessarily immediate descendants
* ol li {
* **color**: red;
* }
* Select all li that are immediate children of ol
* ol **>** li {
* **color**: red;
* }
* Select all input fields with the attribute type=text
* input**[**type**=**text**]** {
* **background-color**: red;
* }
* Select all buttons with the pseudoclass hover
* button**:hover** {
* **background-color**: orange;
* }
  + A ‘pseudoclass’ is a special state of an HTML element. In this example, the state is whether or not the cursor is hovering over a button.
* Select all before pseudoelements of the element a
* a**::before** {
* **content**: "\21d2 Click here: ";
* **font-weight**: bold;
* }
  + A ‘pseudoelement’ is a way to affect certain parts of an HTML element. In this example, the before selector applies content with its included styling before the contents of all a elements.
  + \21d2 is a hexadecimal value for a Unicode icon, which can represent symbols like emoji.
* Select all selection pseudoelements of the element p
* p**::selection** {
* **color**: red;
* **background-color**: yellow;
* }

[**Responsive Design**](https://cs50.harvard.edu/web/notes/1/#responsive-design)

* Responsive design is the idea that a website should look good regardless of the platform its viewed from.
* One way we can do this is by using a ‘media query’:
* **<**style**>**
* **@media** print {
* **.screen-only** {
* **display**: none;
* }
* }
* **</**style**>**
* **<**body**>**
* **<**p class**=**"screen-only"**>**This will not appear when printed**</**p**>**
* **</**body**>**
  + @media is a media query, which means the following CSS will be applied only in certain situations, namely, when the webpage is being printed. .screen-only is a class selector which identifies what content we want to be print only
  + **@media** (min-width: 500px) {
  + body {
  + **background-color**: red;
  + }
  + }
  + **@media** (max-width: 499px) {
  + body {
  + **background-color**: yellow;
  + }
  + }
  + When the width of the screen is at least 500px, the background color of body will be red, while if it is less than 499px, the background color of body will be yellow.
  + In order to interact with the screen size, the following must be included in head: <meta name="viewport" content="width=device-width, initial-scale=1.0">
    - viewport is the visible area on which the screen is being displayed. content refers to the entire webpage the width of which is being set to device-width.
* Another tool is ‘flexbox’. Flexbox allows for the reorganization of content based on the size of the viewport.
* **.container** {
* **display**: flex;
* **flex-wrap**: wrap;
* }
  + By setting display: flex and flex-wrap: wrap, content will wrap vertically if necessary, so no content is lost when the width of the screen is shrunk.
* A grid of content can be achieved in a similar fashion.
* **.grid** {
* **display**: grid;
* grid-column-gap: 20px;
* grid-row-gap: 10px;
* grid-template-columns: 200px 200px auto;
* }
  + By setting display: grid, all the different characteristics of a grid layout can be used to format content. In particular, when defining grid-template-colummns, the final column can be set to auto, filling up however much screen space may be left. If multiple columns are set to auto, they will equally share the remaining space.

[**Bootstrap**](https://cs50.harvard.edu/web/notes/1/#bootstrap)

* Bootstrap is a CSS library written to help make clean, responsive, and nice-looking websites without having to remember the gritty details about flexboxes or grids everytime a layout needs to be set up.
* The only thing needed to use Bootstrap is by adding a single line which links Bootstrap’s CSS stylesheet: <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.1/css/bootstrap.min.css" integrity="sha384-WskhaSGFgHYWDcbwN70/dfYBj47jz9qbsMId/iRN3ewGhXQFZCSftd1LZCfmhktB" crossorigin="anonymous">.
* Bootstrap’s CSS will make everything look a little cleaner and more modern, but its real power comes with its layout system. Bootstrap uses a column-based model where every row in a website is divided into 12 individual columns, and different elements can be alloted a different number of columns to fill.
* Bootstrap’s columns and rows are referenced in HTML with class="row" and class="col-3" attributes, where the number after col- is the number of columns the element should use.
  + Elements can take up a different number of columns based on the size of the screen with attributes like class="col-lg-3 col-sm-6. On a small screen, 6 columns will be used, but in a large screen, 3 columns will be used. If another row has to be added, Bootstrap will do so automatically. This is a much easier alternative to something like flexbox (Bootstrap does so behind the scenes).
* Bootstrap has a whole host of other pretty components which can easily be applied by simply adding the appropriate class attribute to an element. See [Bootstrap’s documentation](https://getbootstrap.com/docs/4.1/components/alerts/) for an extensive list.

[**Sass**](https://cs50.harvard.edu/web/notes/1/#sass)

* Sass is an entirely new language built on top of CSS which gives it a little more power and flexibility when designing CSS stylesheets and allows for the generation of stylesheets in a programmatic way. Ultimately, Sass just makes writing CSS easier.
* In order to use Sass, it must first be [installed](http://sass-lang.com/install). Once installed, we can execute sass style.scss style.css to compile our Sass file style.scss into sass.css, which can actually be linked to and interpreted by an HTML file.
  + If recompiling gets annoying, sass --watch style.scss:style.css to automatically recompile style.scss as style.css whenever style.scss is modified. Additionally, many website deployment systems, like GitHub Pages, have built in support for Sass. For example, if an .scss file is pushed to GitHub, GitHub Pages will compile it automatically.
* One feature of Sass is variables, which are defined as so: $color: red;. Anywhere $color is passed as a value for a CSS property, e.g. color: $color, red will be used.
* Another feature is nesting, which is a more concise way to style elements which are related to other elements in a certain way.
* div {
* **font-size**: 18px;
* p {
* **color**: blue;
* }
* ul {
* **color**: green;
* }
* }
  + In this example, all ps inside divs will be have color: blue, but also font-size: 18px, while uls inside divs will have color: green instead, but still also font-size: 18px.
* One more useful feature is inheritance, which is similar to the object-oriented concept. Sass’s inheritance allows for slight tweaking of a general style for different components.
* **%**message {
* **font-family**: sans-serif;
* **font-size**: 18px;
* **font-weight**: bold;
* }
* **.specificMessage** {
* @extend %message;
* **background-color**: green;
* }
  + %message defines a general pattern that can be inherited in other style definitions using the @extend %message syntax. In addition, other style properties can be added.