RF Bootcamp 2020

Day 3: Reproducibility and Version Control

Review from Day 1

Add these commands as aliases into your `.bashrc` file on the Yen(s):

```
# General Git aliases
alias gs='git status '
alias ga='git add '
alias gb='git branch '
alias gc='git commit '
alias gd='git diff '
alias go='git checkout '

alias got='git '
alias get='git '
```

Today's learning goals

- Understand value of reproducibility
- Learn what dependency management, version control and project management software do
- Learn to use `git` for version control
- Learn to use `Github` for project management

Reproducibility is extremely important in social sciences

- Some fields/journals explicitly require code and data to be submitted so other researchers can re-do your work
- Even if there's no explicit reproducibility requirement, empirical research projects take a long time
  - Receiving data to publishing an article can take **half a decade**
  - Authors may need to modify analysis at any time before publication
    - e.g., do a new analysis after receiving feedback or to respond to a reviewer
- **Plan ahead and adopt processes and technologies to facilitate reproducing old analysis**

Worth a close read...

Gentkow and Shapiro. *Code and Data for the Social Sciences: A Practioner's Guide* (link)

- This presentation draws liberally from this manual

Reproducibility is also self-serving 😄

- **2020:** write some code for Professor X
- **2022:** move across the country and start your PhD
- **2023:** Professor X calls you during finals week...
  - Did you drop all the observations that were missing the *income* variable?
  - Do you remember why we did that?

Invest in making it painless to answer this question (relatively)

Recognizing different levels of reproducibility

- Not a universal standard, just sharing some anecdotes
- How much emphasis you put on reproducibility will depend on your project and the professor you are working with
  - Reproducibility requires effort on the part of **all** collaborators
  - Try your best
- Hopefully, you will adopt this for your own projects and with your own collaborators down the line

Level 1 Reproducibility: *It's all there...*

Most projects are at this stage at some point:

- All the code that you used for the project is in one folder
  - Backed up on Dropbox/Google Drive
- Every graph, table and supplementary analysis is coded up as part of some script
  - One main script that does most of the cleaning and analysis
  - Additional scripts for ad-hoc analysis
- Multiple copies of code, but different copies are timestamped
- Detailed comments on the non-obvious sections of code
- Long google doc that explains what order to run files

Minimally sufficient but...

Level 1 Example

Files in your directory:

```
cleandata_022113.do      cleandata_022613.do       regressions.log
cleandata_022113a.do     cleandata_022613_jms.do   regressions_022413.do
chips.csv                tvdata.dta                regressions_022713_mg.do
regressions_022413.log
```

Source: *Gentkzow and Shapiro*. Code and Data for the Social Sciences: A Practitioner's Guide.

Level 1 Reproducibility: Discussion

- **No guarantee that you can actually re-produce any analysis**:
  - *Example:* you wrote the order of scripts wrong
  - *Example:* `dplyr` goes from v.0.6 to v0.7 and now `summarize_all` works differently
- **Hard to collaborate with other people**
  - *Example:* Someone else adds something to your code and now it doesn't run.
    - What did they add? Why did it break the code?
- **Did you document the right things?**
  - Something seemed obvious at the time; very unobvious 3 years later
- **Deleted a file accidentally**

Level 2 Reproducibility: Manual Pipeline

How could you improve on Level 1?

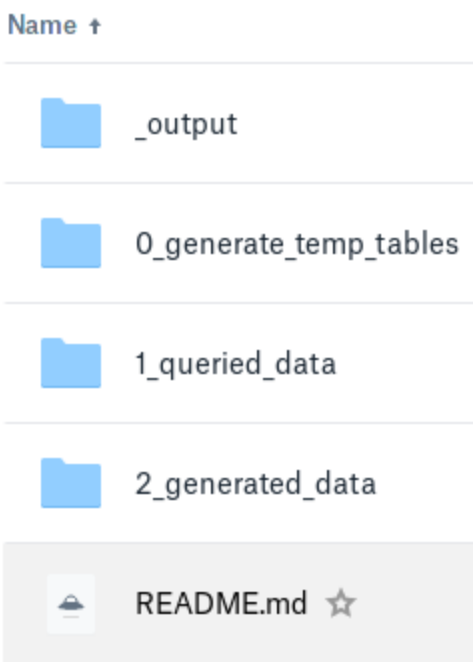
- Build a pipeline of scripts
  - Split up larger scripts into smaller scripts (and sub-folders)
  - Each script has clear **inputs** and **outputs**
    - These outputs can be inspected
  - One "wrapper" script to run the "pipeline" (all the scripts in order)
- Only one copy of each script at a time
  - Put all the older versions in a `/old/` folder
- Choose and maintain a systematic documentation style, e.g.,
  - Every script has a comment explaining **purpose**, **dependencies**, **expected output**
  - Every folder has its own README file explaining **purpose**, **dependencies**, **expected output**

Level 2 Reproducibility: Examples

```
---C:/build---
/input
    extractOB.xls

/code
    rundirectory.bat
    export_to_csv.stc
    mergefiles.do

/output
    tvdata.dta
```

```
---C:/analysis---
/input
    tvdata.dta (link to C:/build/output)

/code
    rundirectory.bat
    regressions.do
    regressions_alt.do

/output
    fig1.eps
    fig2.eps
    tables.txt
```

Source: *Gentkzow and Shapiro*. Code and Data for the Social Sciences: A Practitioner's Guide.



Source: My last project

Level 2 Reproducibility: Discussion

- We know what order to run scripts in
  - **But, still no guarantee that the scripts will actually run if versions change**
- Systematic documentation means that we "over-comment" how everything works
  - Over-commenting is almost always appreciated by new and returning readers
  - **While we've documented what things to do, we still haven't documented *why* we did things**
- Pipeline means we can isolate what parts of a script break when a collaborator adds something and fix it faster
  - **However, we still have to figure out what they added**
  - Sometimes, you're just trying to retrace your own steps from 6 months ago

How to improve on level 2?

In addition to structuring your code and data as a pipeline:

- Use **dependency management** software, if available
  - Keeps track of the version of software that you depend on
  - Depends on what language you are using, so will point to common implementations
  - Otherwise, try to document what software versions you use
- Use **version control** software
  - Software to track file changes
  - Will cover this extensively today
- Use a **project management system**
  - Fancy "to-do" list

Dependency Management

Dependency Management

Two goals:

1. Document what versions of software you are using
2. Make it easy to install the same versions of software on another machine

---

- Collaborator's machine might have a different version of `Stata`, different version of an `R` package, etc.
  - Make sure they can run your code on their machine
- Whether this is easy or hard depends on the language

Dependency Management: Stata

- I've never done this in Stata
  - A lot of things can be done without installing modules, so you may just have to document what version of Stata you are using
- If you are using modules, maybe check out Haghish's Github Module

Dependency Management: Python (Basic)

- By convention, modules that are required are listed in a text file called `requirements.txt`
  - Each row specifies a module and a minimum (or exact) version number

```
# example requirements.txt
matplotlib==2.3.3
pandas>=1.0
```

- You can install all required modules at once by running in shell:

```
pip install -r requirements.txt
```

- You can automatically create a requirements file by running in shell:

```
pip freeze > requirements.txt
```

Dependency Management: Python (Advanced)

- Use virtual environments to emulate a "clean" Python install
  - Install only the modules that are needed
  - Run multiple projects that need conflicting package versions on the same computer
- Three ways to make a virtual environment
  - `venv` (built-in to Python 3.3+)
  - `virtualenv` (package with more features)
  - Conda environments

Dependency Management: R

- Install the `packrat` package

```
install.packages("packrat")
```

- Simplified way to create a "snapshot" of all the packages that are installed on your system
- Whenever you want to freeze the package versions, call inside R:

```
packrat::snapshot()
```

- Creates a `packrat.lock` file that records metadata of all the packages in your snapshot
  - If using only publicly available packages, this would be sufficient documentation
- Will automatically restore package versions back to the current snapshot in a R project
- For more detail, see this walkthrough

Version Control

What is a version control system?

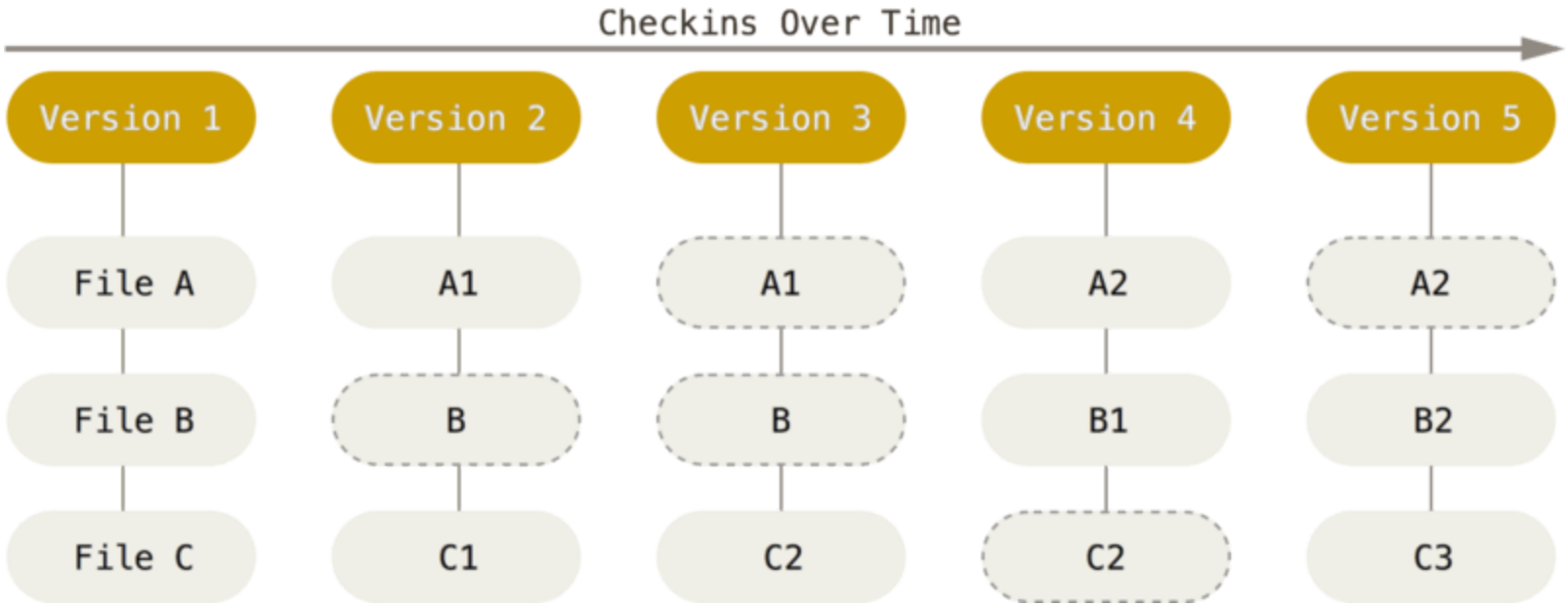
> Version control systems are a category of software tools that help a software team manage changes to source code over time. Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

Source: Bitbucket

- TL;DR Software that keeps track of how files change
  - Makes it easier to work with multiple versions of the same file

Under the hood, `git` keeps track of snapshots

Source: git-scm.com



- Snapshot 2 has changed A and C, Snapshot 3 has changed C only, etc.
- Stores one copy of each **version** of a file
  - Be careful with storing data in git

Glossary

- **Repository** - a folder, with the history of all of its files
- **Commit** - snapshot of your folder at a given point in time

General workflow:

1. Make changes to your files
2. **Stage:** Decide which changes you want to keep
3. **Commit:** Make a snapshot of your files
4. **Sync:** Transmit snapshots of your files to the cloud

Working with `git` on Yen(s)

Learn Your Hands

- Log into a Yen and follow-along

Setup connection to Github

1. Generate an SSH key

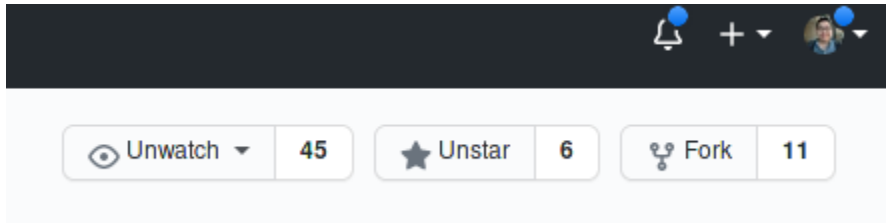
```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_rsa
cat ~/.ssh/id_rsa.pub
```

2. Copy your public key (the output of your last command)
3. Go to https://github.com/settings/ssh/new
   i. **Title:** Yen
   ii. **Key:** Paste your public key in the "key" section
4. Set your default editor to Nano

```
git config --global core.editor nano
```

Go to https://github.com/zhangchuck/my_first_repo

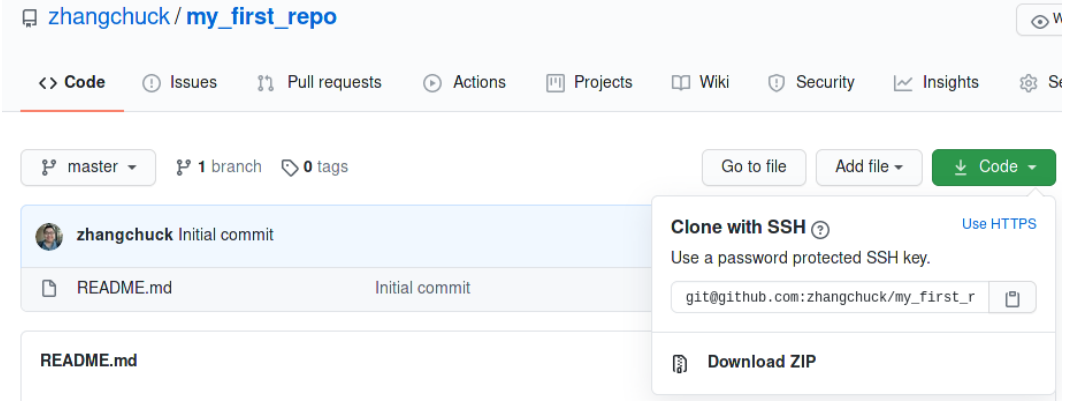
- "Fork" this repository to your account



- You are making a personal copy of all the code
- You will be able to see it online at: `https://www.github.com/[GH_user]/my_first_repo`

```
git clone git@github.com:[GH_user]/my_first_repo.git
```

- Replace [GH_user] with your github username
- "Cloning" a repository means you are making a local copy of a repository (i.e., on Yen)
  - You can find the path for cloning on Github



  - The cloned repository is saved locally
  - Your local repository is linked to a remote repository on Github
    - **NOTE:** Changes you make locally are **not** automatically updated on Github

`cd my_first_repo; lh`

- Only things inside this folder are part of this repository
- Currently just one folder (`.git`) and one file (`README.md`) in this repository
  - Don't touch `.git`, contains all of the history

`nano README.md`

- Make some changes to `README.md`
  - Delete a line
  - Add a couple lines
  - Where was the last place you went on vacation?

`git status`

```
cyzhang@yen4:~/my_first_repo$ git status
HEAD detached at 58fe147
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

- `git status` will tell you which files in the repository have changed since the last commit
  - New files?
  - Modified files?
  - Removed files?
  - Moved files?

`git diff`

- `git diff` tells you which **lines** have changed since the last commit
  - `+` indicates an added line
  - `-` indicates a deleted line
  - Modifications will usually be recored as a `+` and a `-`
- Good way to examine how files have changed
- Can do this all the way back to the first time a file is created!

```
nano NEWFILE.md
```

- Create a new file
- Add some content
- Save it

`git status; git diff`

- Note that `NEWFILE.md` is listed as "untracked"
  - You haven't told the repository to keep track of changes to `NEWFILE.md` yet
  - Doesn't show up in `git diff`
- `README.md` is tracked
  - `git` tells you that it's been modified
  - `git diff` will tell you how it's been changed

`git add README.md`

```
cyzhang@yen4:~/my_first_repo$ git status
HEAD detached at 58fe147
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        NEWFILE.md

nothing added to commit but untracked files present (use "git add" to track)
```

- This adds the changes to `README.md` to "staging"
  - The step before committing
  - Allows you to group changes to a bunch of files together
  - Changes haven't been saved yet

```
git commit -m "My first commit"
```

- Commits the changes that have been staged to the repository
- You **must** include a message with every commit
  - Usually, try to describe what has changed or why you are changing things

Source: xkcd

`ga *`

- `ga` we added this alias at the beginning of this session
- `*` means everything in the directory
  - Wildcard for multiple files we discussed last week
- What will be "staged"?
  - Just `NEWFILE.md`
  - No changes to any other file

`gc -m "Adding a new file"`

- `gc` we added this alias at the beginning of the session
  - `git commit`

Check your repo online

https://github.com/[GH_user]/my_first_repo

- You won't see any changes
- Changes have been committed (added) to your local repository
  - Local repository needs to be synced with the remote repository

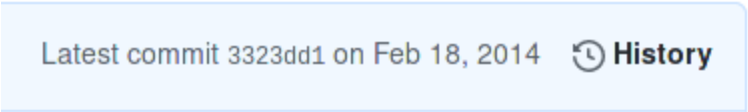
`git push`

- This will *try* to add your new commits to the remote repository
- Why might it fail?
  - Someone else made changes to the code (handled by *auto-merge*)
  - Someone else made changes to the **same parts** of code that you did!
- Process for de-conflicting commits is called "merging"
  - Makes use of `git diff` to point out exactly why commits conflict
- Right now, you're the only contributor to this repo, so not an issue

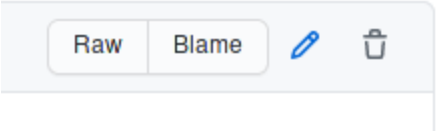
Check your repository online

https://github.com/[GH_user]/my_first_repo

- Your changes will now be registered
- Click on `README.md`
- You can look at the history of your file



- Go back, and use the online editor to make a change to the file and commit it



- Remote repository (on Github) is now 1 commit **ahead** of your local repository

`git pull`

- `git pull` syncs your local repository with the remote repository
  - Will fail if there are any conflicts

`git log`

- Shows you the history of commits in your local repository
- Each commit is identified by a `commit id`

```
commit 29708f6e5a3c4e2c599707cc5866b7d5f0e5f215
Author: Charles Zhang <zhangchuck@users.noreply.github.com>
Date:   Sun Feb 9 00:44:45 2020 -0800

    Clean up whitespace
```

`git checkout [commit id]`

- You can "rewind" your entire repository to the state when `commit id` was committed
- Try the using the commit id of the **initial commit**
  - `58fe14713b42b042d382781068e96b35035399f7`
- Examine the folder
  - `NEWFILE.md` is no longer there
  - `README.md` has been reverted to an older state

```
git checkout master
```

- Return to the newest commit in the **master branch**

`git rm README.md`

- Deletes a file *and* stops it from being tracked

```
cyzhang@yen4:~/my_first_repo$ git status
On branch feature#second_test
Your branch is up to date with 'origin/feature#second_test'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    README.md
```

- If you just `rm` a file, then you still need to tell git that you want to stop trackinng the file:

```
cyzhang@yen4:~/my_first_repo$ git status
On branch feature#second_test
Your branch is up to date with 'origin/feature#second_test'.

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    README.md
```

`git mv NEWFILE.md OLDFILE.md`

- Renames/moves a file in a way that `git` understands:

```
cyzhang@yen4:~/my_first_repo$ git status
On branch feature#second_test
Your branch is ahead of 'origin/feature#second_test' by 1 commit.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    NEWFILE.md -> OLDFILE.md
```

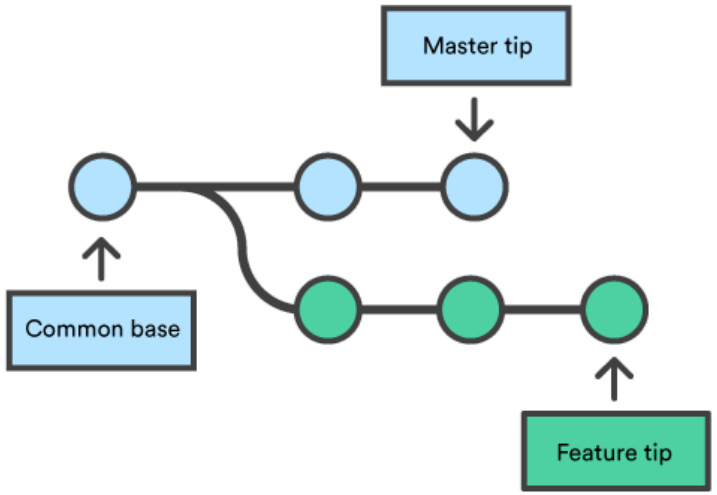
- If you just `mv` the file (e.g., `mv NEWFILE.md OLDFILE.md`), then `git` since you deleted `NEWFILE.md` and created a new, untracked file `OLDFILE.md`:

```
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    NEWFILE.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        OLDFILE.md
```

`git reset --hard`

- This is a bit dangerous...
- Resets the state of your folder to the last **commit**
- **Deletes any work you have done since the last commit**
  - Great if you made a bunch of mistakes or tried something out that didn't work

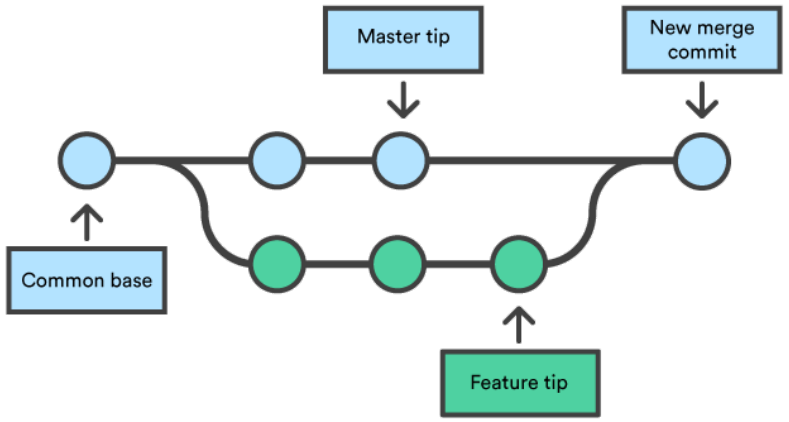
New Concept: (Feature) Branches



Source: *Bitbucket*

- Each circle is a **commit**
- Arrows represent relationship between commits
- Sequence of light blue commits is the **Master** branch
  - What we've been working on so far
- Sequence of green commits is a "feature branch"
  - Starts from a commit in the master branch
  - Use it to run experiments

Merge feature branches back into master



Source: *Bitbucket*

- You can merge feature branch back into MASTER
- Two people can safely work on two separate branches
  - Resolve conflicts later
- Rule-of-thumb: code on master should be **functioning** and represent a stable/fully-working version of your code

`git branch`

- List all the branches that exist
- Right now, only *master*

```
git branch feature#first_test
```

- Create a **new branch** called feature#first_test

`git branch`

```
cyzhang@yen4:~/my_first_repo$ git branch
  feature#first_test
* master
```

- You've created a new branch
- You're still *working* in the master branch

```
git checkout feature#first_test
```

```
cyzhang@yen4:~/my_first_repo$ git branch
* feature#first_test
  master
```

- Now, all changes you make well be in the feature#first_test branch

```
git checkout master
```

- Return to the master branch

```
git branch -d feature#first_test
```

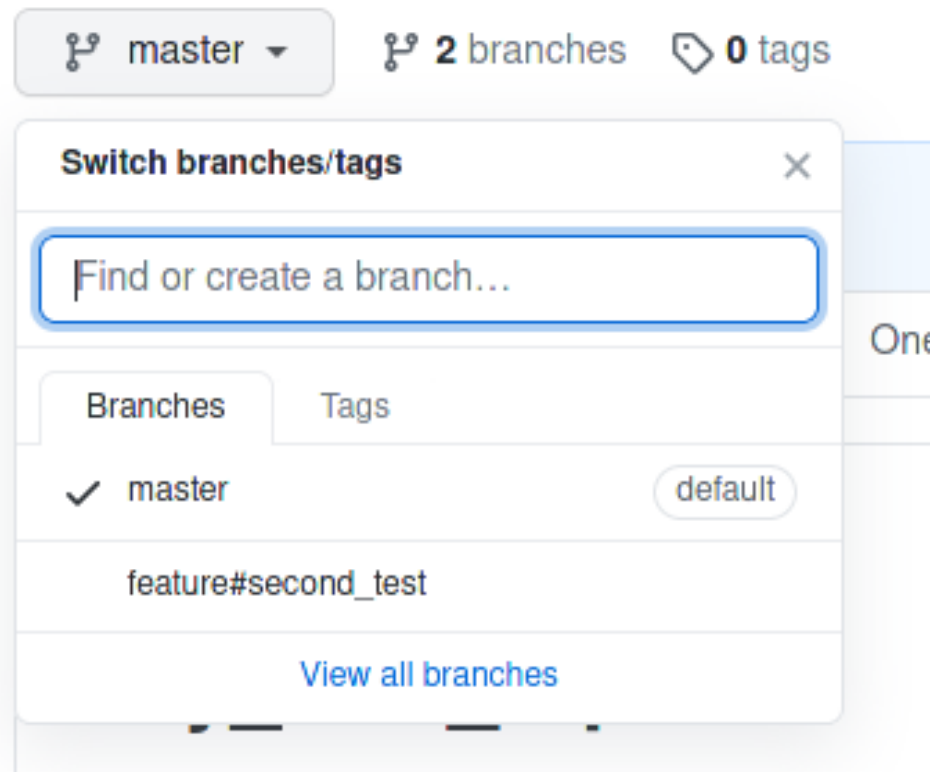
- `-d` option is to delete a branch (be careful!)
- You can't delete a branch when you have the branch checked out
  - That's why we returned to *master*

```
git checkout -b feature#second_test
```

- Convenience: create a new branch `-b` and check it out immediately

Go to your repo on Github again and look at the available branches

https://github.com/[GH_user]/my_first_repo



- The branches don't exist on Github yet!
- Only exist in local repository at the

```
git add *; git commit -m "Just some more changes"
```

- Stage and commit those changes
- These changes are in the *feature#second_test* branch

`git push`

- You will not be able to push and will get this error:

```
fatal: The current branch feature#second_test has no upstream branch.
```

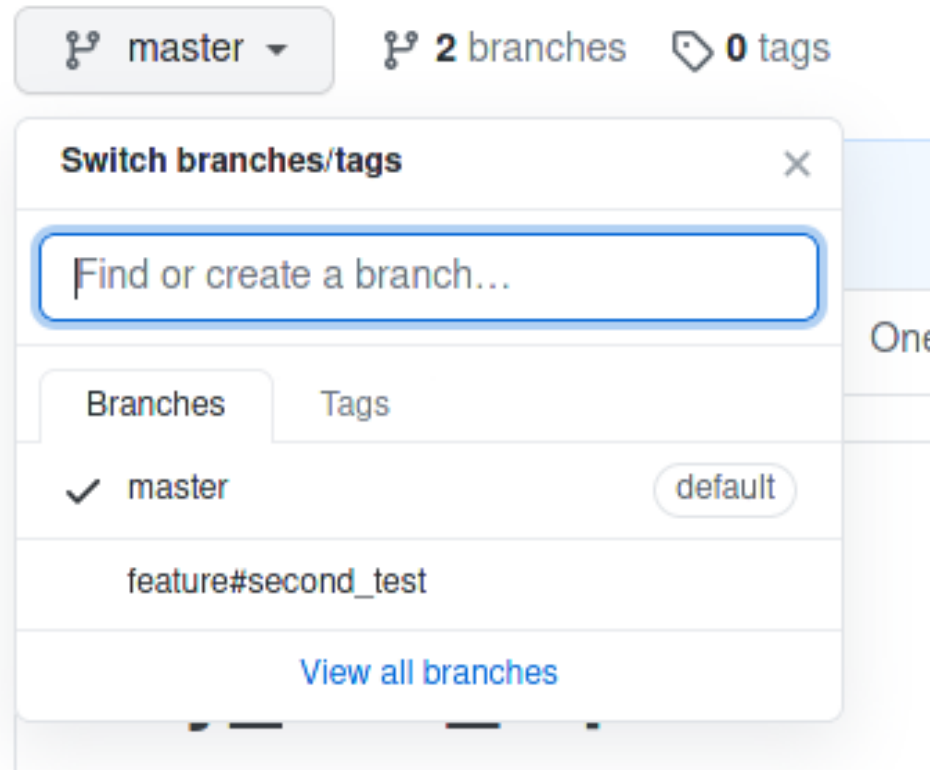
- Your new branch doesn't exist on the remote repository
  - `git` doesn't know where to add the changes

```
git push --set-upstream origin feature#second_test
```

- Do what the error mesage suggests
- "Origin" refers to Github
- This creates a new branch in the Github repository with the name "feature#second_test" and associates it with your local branch

Go to your repo on Github again and look at the available branches

https://github.com/[GH_user]/my_first_repo



- The branches exists now

To do merges, use the GUI on Github

- You can of course manually merge use `git merge` or `git rebase`
  - A lot of the value in merges is discussing big batches of changes
  - "Pull request" process helps with that
- Pull request:
  - Will analyze the differences between the feature branch and the master branch
  - Will highlight any potential conflicts
  - Allows you to discuss things that need to be fixed before merging in
- You can keep on editing the branch until it's merged without creating a new pull request
- Watch my screen

We're done with this repo!

Git: Additional References & Concepts

There are still some other important concepts you'll want to learn if you use `git` / `github` collaboratively:

- Using `.gitignore` to ignore files
  - Especially data files
- Pull Requests
- Feature Branch workflow / `git branch`
- Forking workflow
  - Often seen in open-source projects where they may be many contributors

Git Cleanup: Quality of Life

When you're on your own machine (not on Yen), you can also use GUIs for Git/Github, e.g.

- Github Desktop
- GitKraken

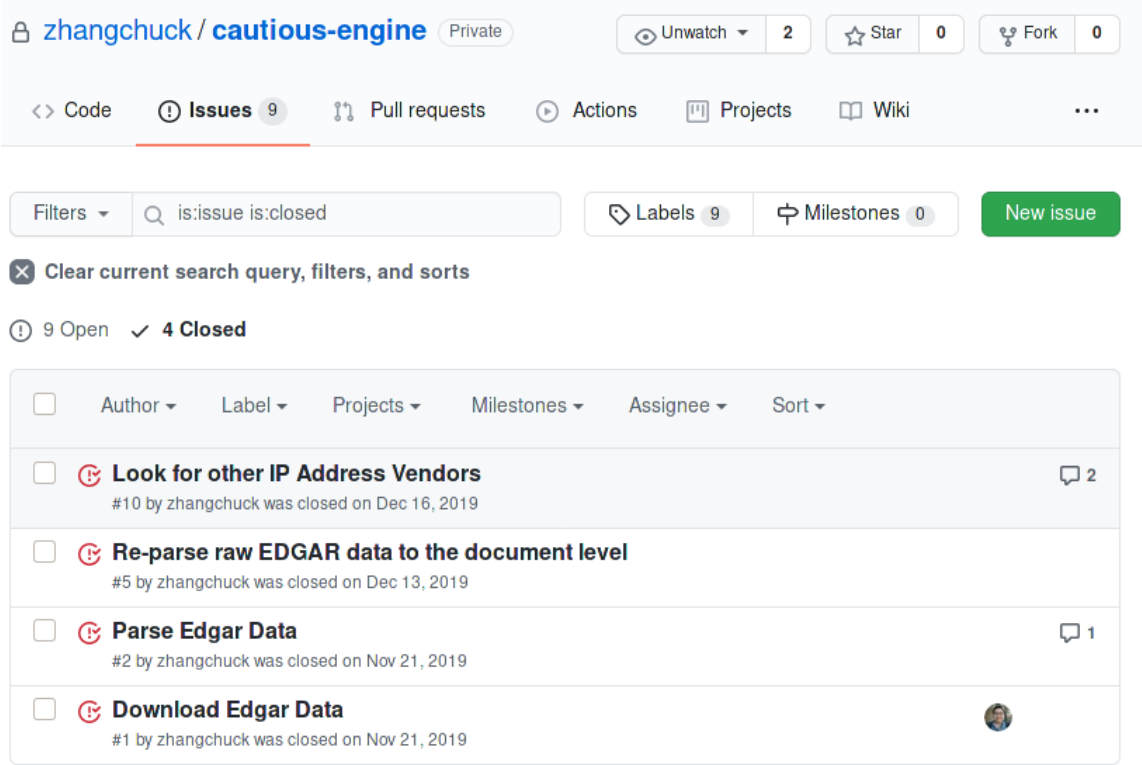
It's also directly integrated into:

- RStudio
- JupyterLab (extension)
- VS Code

Project Management System

Depends on who you're working with

- In general, don't rely on email to keep track of tasks
- Trello / Jira often used (maybe overkill?)
- Another option: every GitHub repository comes with its own "Issues" tracker



Example: GSLab publicly posted project management rules

- Projects
- Tasks
- Sprints

Example: Repo for the EDGAR Data task

- Link
- Not the best example

Exercise

Exercise: Clean EDGAR Data

Still working with the EDGAR Log data from yesterday

- Description / Codebook
- Task is based off of work for *Barrios et al.*, 2020. Informing Entrepreneurs: Public Corporate Disclosure and New Business Formation
  - Jung Ho Choi and Jinhwan Kim are Stanford accounting professors 😄
- Real project analyzed 15 years of data on who accesses public financial information.
  - We're going to work with 3 **days** worth of data but for now, let's focus on the 1 day of data we downloaded yesterday
    - Path is `~/rf_bootcamp_exercise_1/data/log20170615.csv`

First setup your project environment

1. Create a new repository `[GHuser]_edgar_exercise` on Github
2. Clone it onto the Yens

Write and run (on Yen) a script to do some basic cleaning of the data

Script should:

1. Remove all crawlers
2. Construct a dataset that describes **how often** a document related to a company was accessed by a given user on a given day
   - i.e., the dataset should be at the ip-day-company-document level
     a. Users are assumed uniquely identified by `ip`
     b. Companies are uniquely identified by `cik`
     c. Document is uniquely identified by `accession`
3. Save this file to `~/rf_bootcamp_exercise_1/data/log20170615_cleaned.csv`

Use the language (Stata/Python/R) you are most comfortable with and please comment liberally

Commit this code to your repo

- Note: do **not** commit the data to your repo
  - Repos are not great at dealing with large files
    - Store code, not data
    - Use `.gitignore` https://www.jstor.org/stable/43193723?seq=1#page_scan_tab_contents

Write a separate script that constructs a histogram using the cleaned data

- Histogram of the # of times a document was accessed
  - Y-axis should be # documents
  - X-axis should be # of times accessed
- Save this graph in your repo
- Commit the graph and your code to your repo

Use any language you'd like (doesn't have to be the same as the cleaning script)

Reorganize your repo

- Is there a logical separation of scripts? Folders?
  - If you need to make adjustments, make them and make another commit

Send me the link to the Github page with your code and you're finished!