ADACS ASTRO HACK WEEK 2020

# INTRO TO VERSION CONTROL

# WHY USE VERSION CONTROL?

- Back up your project!

# WHY USE VERSION CONTROL?

- Back up your project!
- Keep track of changes that have been made

# WHY USE VERSION CONTROL?

- Back up your project!
- Keep track of changes that have been made
- Tag stable versions, versions used in particular analyses
  - good for reproducibility

# WHY USE VERSION CONTROL?

- Back up your project!
- Keep track of changes that have been made
- Tag stable versions, versions used in particular analyses
  - good for reproducibility
- Code distribution

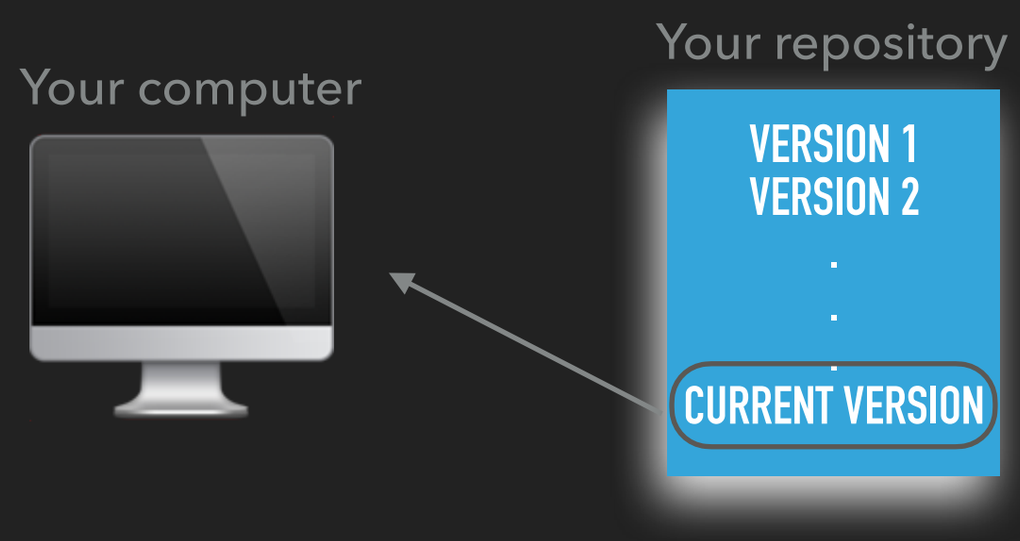
# WHY USE VERSION CONTROL?

- Back up your project!
- Keep track of changes that have been made
- Tag stable versions, versions used in particular analyses
  - good for reproducibility
- Code distribution
- Easy collaboration
  - Great for things like writing papers

# AVAILABLE TOOLS

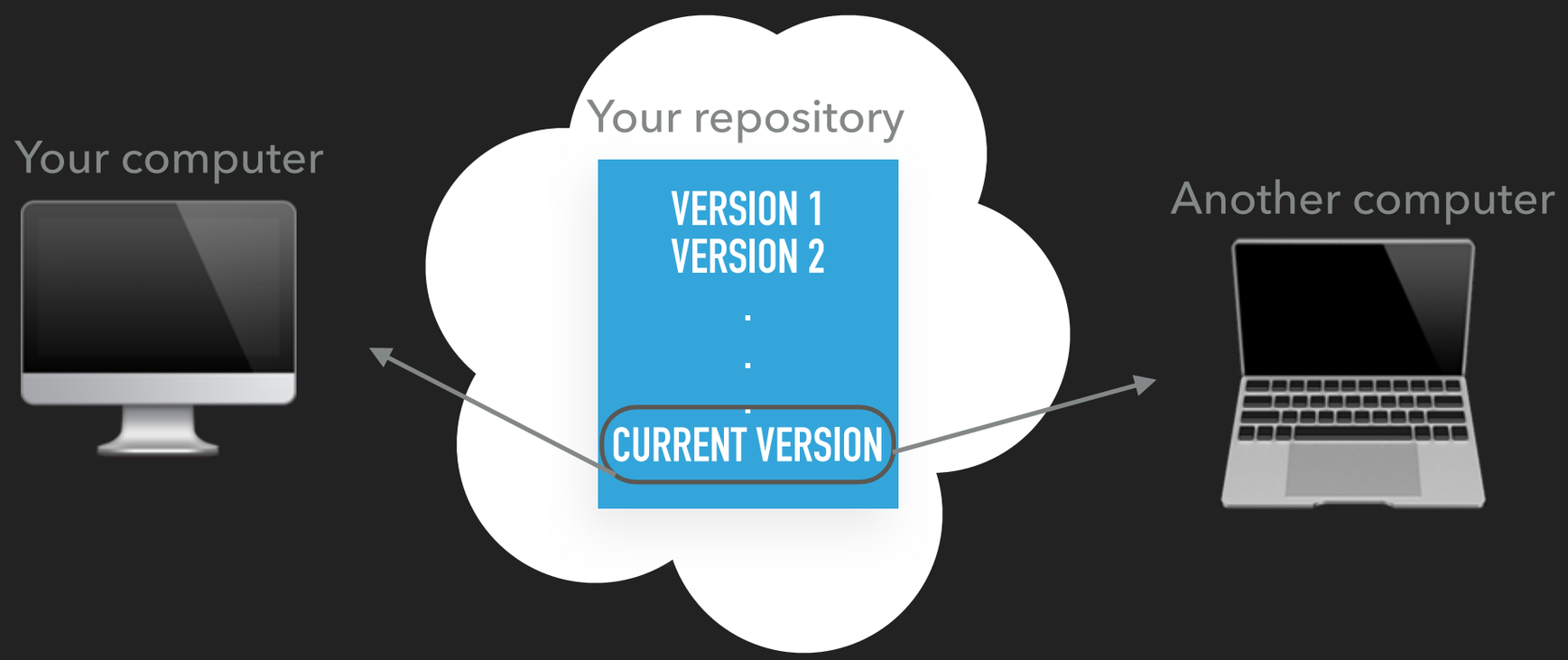
- **Git**
  - Most common now
- Mercurial
- SVN

# REPOSITORY




Repository can be: **Local** (on your own computer)

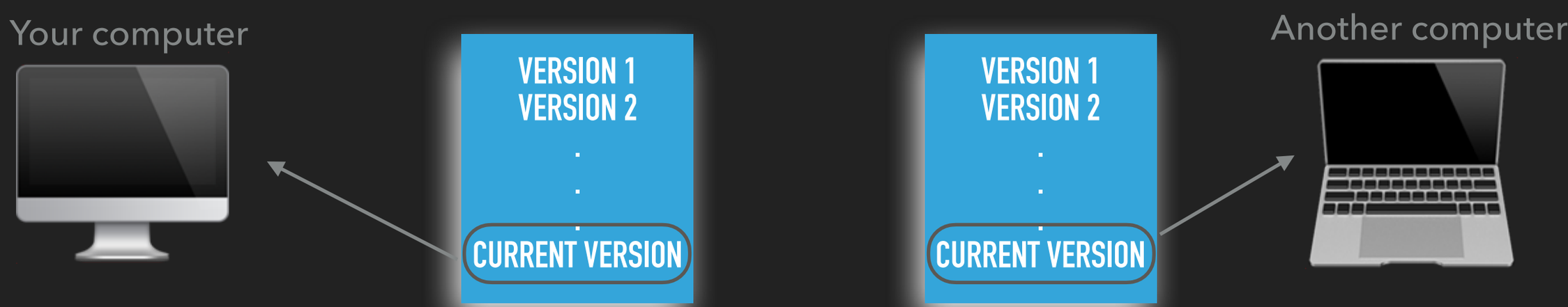
# REPOSITORY




Repository can be: **Centralized** (Stored remotely, you only have the current version on local computer)

(Example: SVN)

# REPOSITORY


Repository can be: **Distributed** (Repository on every computer)

*This is how GitHub works*

# MAKING A GIT REPOSITORY

```
$ cd <project directory>
$ git init
```

Think carefully about what goes in each repository

- Generally one repository per project
- (i.e. don't make repository for your whole desktop!)
- Don't nest repositories!

# HOW DO I KNOW IT WORKED?

- Directories under version control will have a **.git** directory

```
$ ls -a
```

- The **.git** directory has all of the info about change history of your project
  - Generally a good idea to ***not*** mess around in that directory yourself
- Can also check with:

```
$ git status
```

```
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

# DO IT YOURSELF: MAKE YOUR FIRST REPOSITORY

```
$ mkdir my_first_repository
```

```
$ cd my_first_repository
```

```
$ git init
```

```
$ ls -a
```

```
$ git status
```

# DO IT YOURSELF: MAKE YOUR FIRST REPOSITORY

Make a file in your directory:

```
$ echo "this is the first line of text" > my_first_file.txt
```

(can also use your favorite text editor)

```
$ git status
```

What's the output now?

# ADDING THINGS TO YOUR REPOSITORY

```
$ git status
```

```
On branch master

No commits yet

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

   my_fist_file.txt

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

Need to specifically tell git which files to track

```
$ git add <filename>
```

# ADDING THINGS TO YOUR REPOSITORY

`git add` stages your commit

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

   new file:   my_fist_file.txt
```

I.e. tells git that you're ready to add this version of your file to the repository

# NOW IT'S TIME TO COMMIT!

```
$ git commit -m "my first commit"
```

"-m" flag is followed by a description of changes made in this commit

(if you don't use "-m" you'll be prompted to add a commit message)

A version of your project now lives in the repository!

# DO IT YOURSELF: STAGE AND COMMIT

```
$ git add my_first_file.txt
```

```
$ git status
```

What's the output now?

```
$ git commit -m "my first commit"
```

# RECAP AND TIPS

- Edit your code/paper/etc
- *Stage* your commit with git add
  - `$ git add <filename>`
  - `$ git add -u` will stage all of the files currently being tracked
  - Avoid using `$ git add *`!!
- *Commit* your changes

# THINGS TO NOT ADD TO YOUR REPOSITORY

- *LARGE FILES*
- Raw data
- Files that are automatically generated

Use a `.gitignore` file

Example file:

```
a_specific_file.txt
*.dat (e.g. all data products)
an_entire_directory/
```

# SOME COMMON ISSUES AND HOW TO FIX THEM

# EXAMPLE 1: I EDITED A FILE BETWEEN STAGING AND COMMITTING!

- Every time you edit a file you need to re-stage (i.e. run git add again)
- Can check whether file has been modified with git status!

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

   modified:   my_first_file.txt

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:   my_first_file.txt
```

# EXAMPLE 2: I ADDED SOMETHING I DIDN'T WANT TO ADD!

```
$ rm <filename>
```

# EXAMPLE 2: I ADDED SOMETHING I DIDN'T WANT TO ADD!

```
$ rm <filename>
```

Not what you want!!

Does not remove the file from git

# EXAMPLE 2: I ADDED SOMETHING I DIDN'T WANT TO ADD!

```
$ rm <filename>
```

Not what you want!!

Does not remove the file from git

```
$ git rm <filename>
```

Remove the file from repository

```
$ git rm --cached <filename>
```

Don't delete it, but stop tracking it

# EXAMPLE 3: I CAN'T REMEMBER EXACTLY WHAT I CHANGED

```
$ git diff
```

```
diff --git a/my_first_file.txt b/my_first_file.txt
index 9f26126..a81af58 100644
--- a/my_fist_file.txt
+++ b/my_fist_file.txt
@@ -1,2 +1,3 @@
 This is the first line!
+this is the second line.
```

Tells you what changes you've made since last commit

# DO IT YOURSELF: USING GIT DIFF

```
$ echo “a second line of text" >> my_first_file.txt
```

(or use your favorite text editor)

```
$ git diff
```

What’s the output?

# EXAMPLE 3: I WANT TO GO BACK TO AN OLD VERSION OF THE CODE

**Case 1:** You made some edits to a file, realize they're garbage, want to got back to most recent version in repo

```
$ git checkout <filename>
```

(Can think of it as pressing ctrl+Z a bunch of times)

# EXAMPLE 3: I WANT TO GO BACK TO AN OLD VERSION OF THE CODE

**Case 2:** You want to go back several commits

```
$ git log
```

```
commit 947fb8e267dcd26f5bf2f2b885185a1e9eed2ed3
Author: megmillhouse <meg.millhouse@gmail.com>
Date:   Wed Feb 5 20:47:48 2020 +1100

    another new commit

commit eeb65b7ec25fe49f144037cec539affe25a5ef97
Author: megmillhouse <meg.millhouse@gmail.com>
Date:   Tue Feb 4 13:06:15 2020 +1100

    second commit

commit ce38b85cec0a2479853487b933ba5a18ce52ade1
Author: megmillhouse <meg.millhouse@gmail.com>
Date:   Tue Feb 4 12:45:50 2020 +1100

    first commit
```

Each commit gets a unique identifier called a *hash*

```
$ git checkout eeb65b7 <filename>
```

# EXAMPLE 3: I WANT TO GO BACK TO AN OLD VERSION OF THE CODE

**WATCHOUT!**

```
$ git checkout eeb65b7
```

What happens if you don't specify a file?

End up in a "Detached HEAD" state

Fix via:

```
$ git checkout master
```

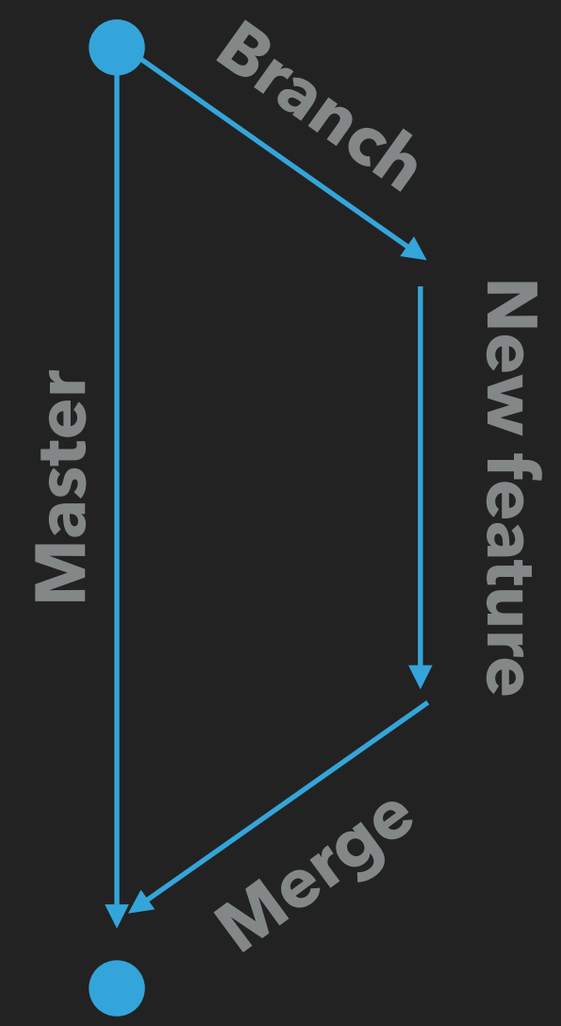
# PROJECT MANAGEMENT: BRANCHING

- For major developments, work on a separate *branch*

```
$ git branch <your-new-branch>
$ git checkout <your-new-branch>
```

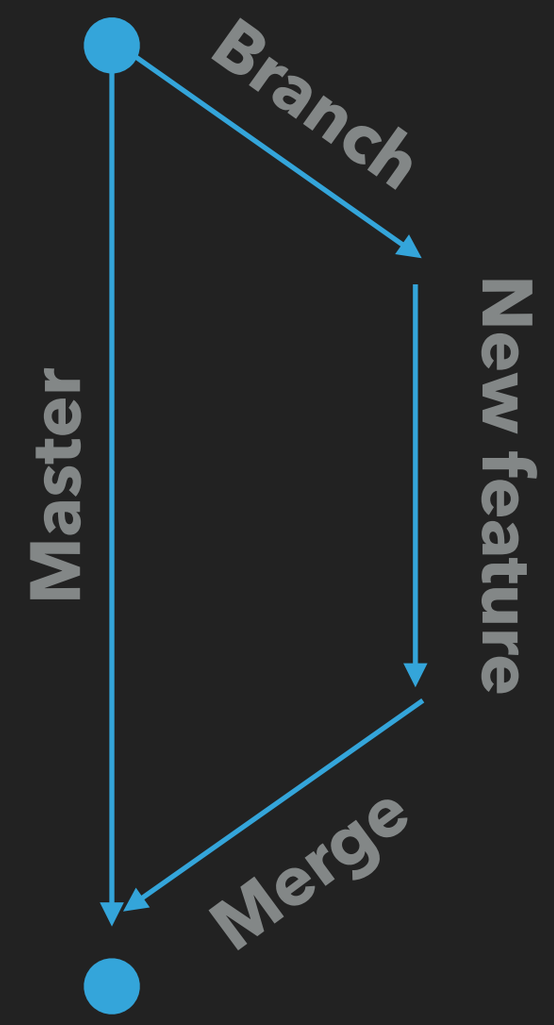
```
$ git checkout master
```

```
$ git merge <your-new-branch>
```



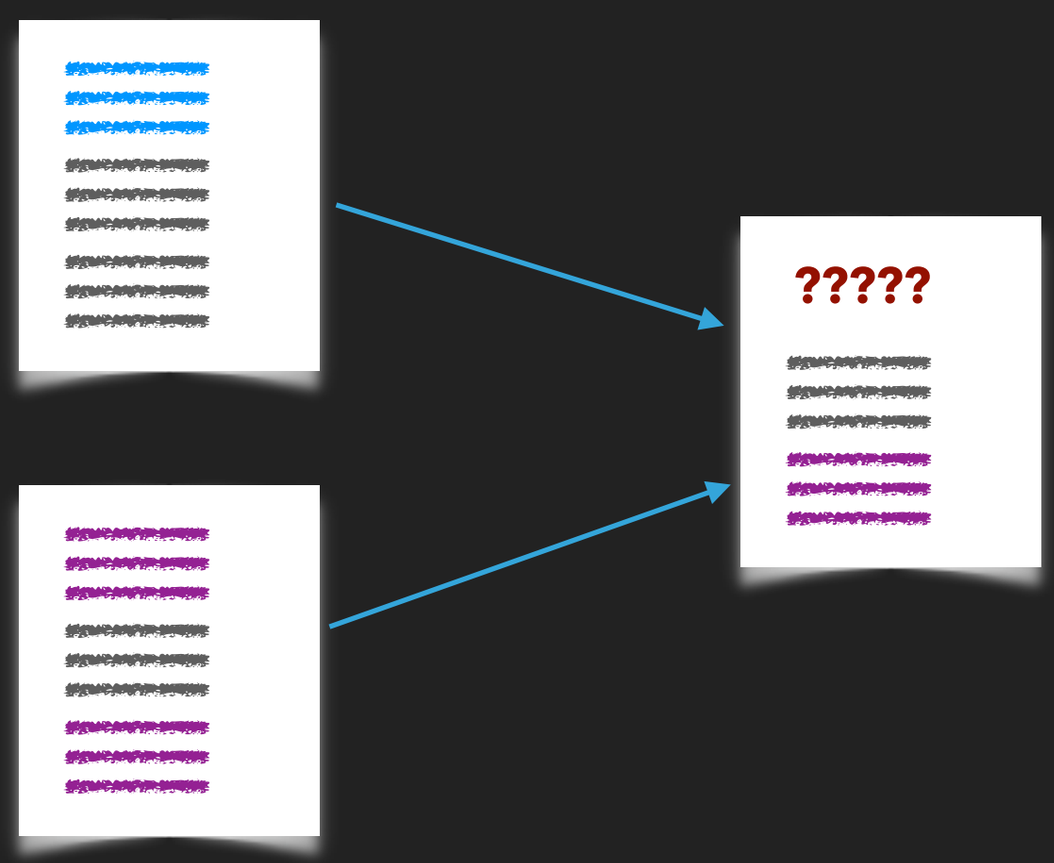
# PROJECT MANAGEMENT: BRANCHING

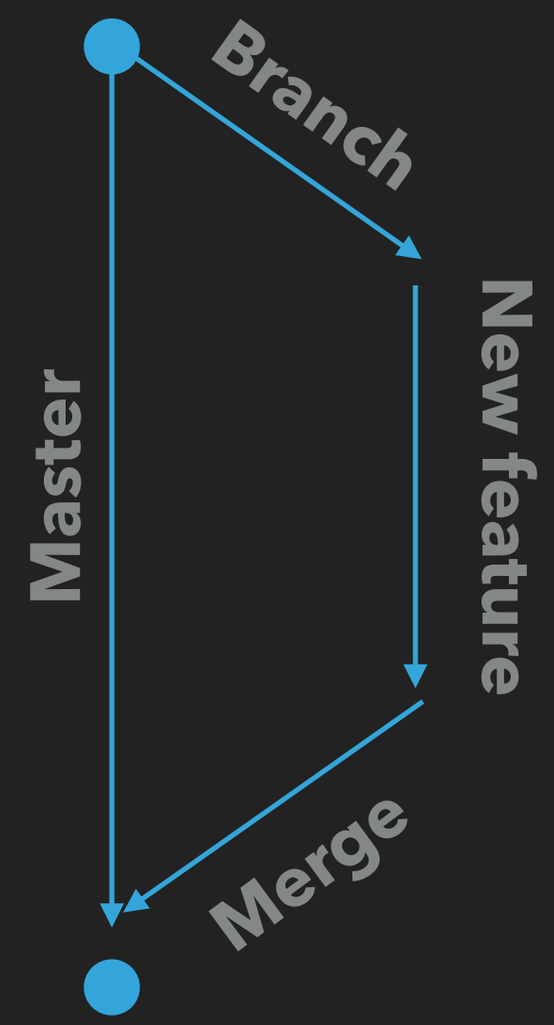
- Can keep working on new branch and master at the same time
- Git is good at managing changes itself!



# PROJECT MANAGEMENT: BRANCHING

- Can keep working on new branch and master at the same time
- Git is good at managing changes itself!



# CONFLICTS

- Conflicts arise when the same line of text has been edited in two branches

You will get an error message like:

```
Auto-merging my_fist_file.txt
CONFLICT (content): Merge conflict in my_fist_file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

And the conflicted file will look like:

```
This is the first line
A second line
<<<<<<< HEAD
This is the third line
=======
Or maybe THIS is the third line
>>>>>>> new_branch
```

# DO IT YOURSELF: BRANCH AND MERGE (WITHOUT CONFLICT)

```
$ git branch a_new_branch
$ git checkout a_new_branch
```

```
$ echo “new text from a branch" >> my_first_file.txt
```

(or use your favorite text editor)

```
$ git checkout master
```

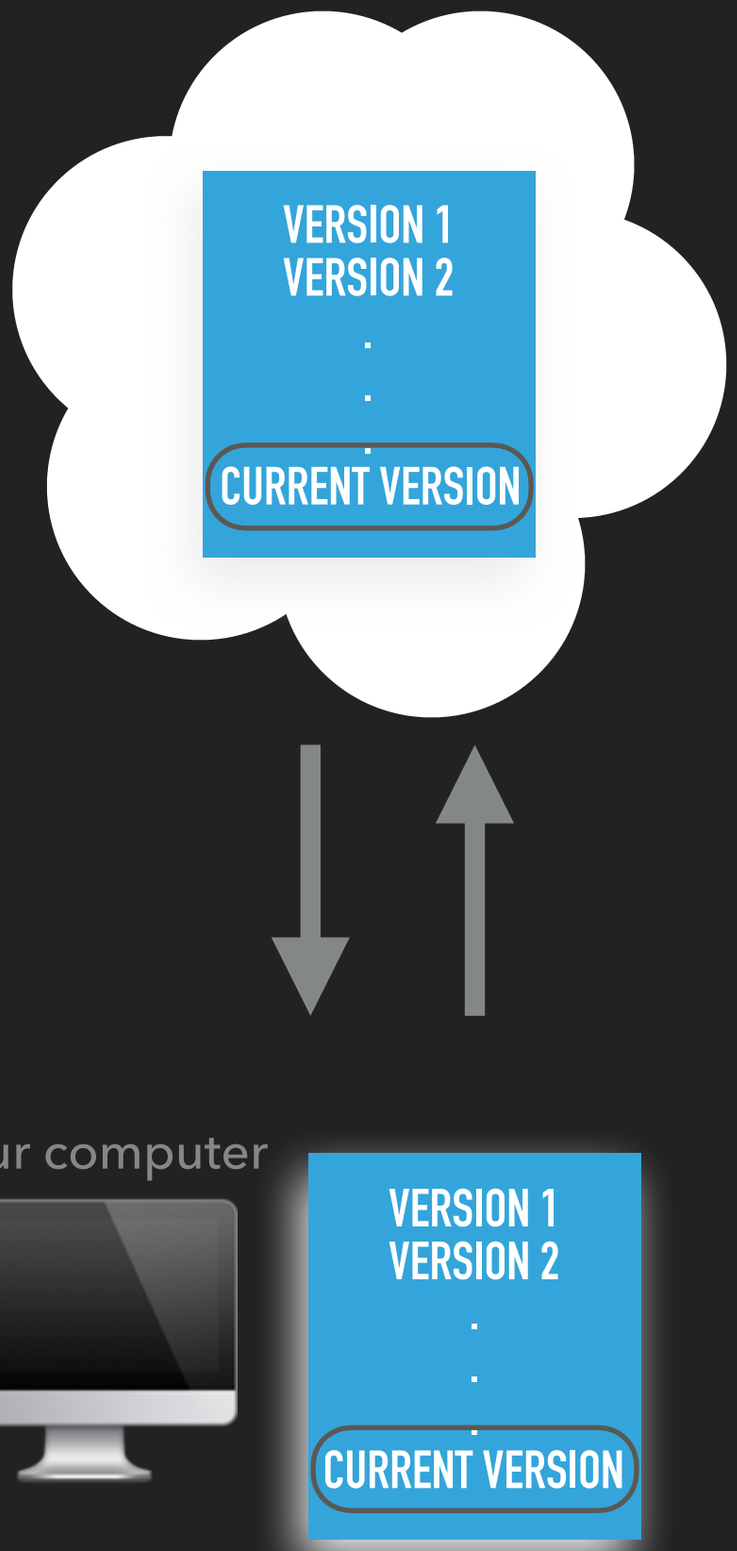
```
$ git merge a_new_branch
```

```
$ git branch -d a_new_branch
```

(delete branch for housekeeping)

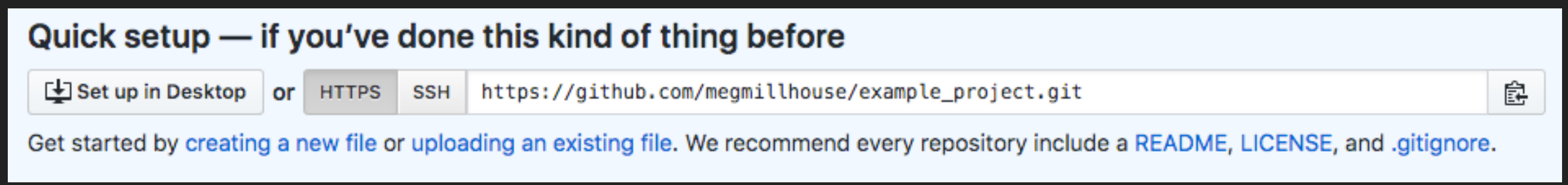
# PUSHING TO REMOTE REPOSITORIES

- Staging and committing only updates the local repository on your own computer
  - Want to edit your code on a different computer
  - Want to share your code with collaborators
- Use remote repository hosting like GitHub, GitLab, Bitbucket



# SETTING UP A REMOTE REPOSITORY

https://github.com/new




```
$ git remote add origin <url>
```

```
$ git push --set-upstream origin master
```

Only need this part once

# PUSHING TO REMOTE REPOSITORIES

Send your current repository to remote location:

```
$ git push origin master
```

Local

Remote

Get the most recent remote repository onto your computer:

```
$ git pull
```

# GOOD HABIT OF DAILY WORK FLOW:

```
$ git pull
```

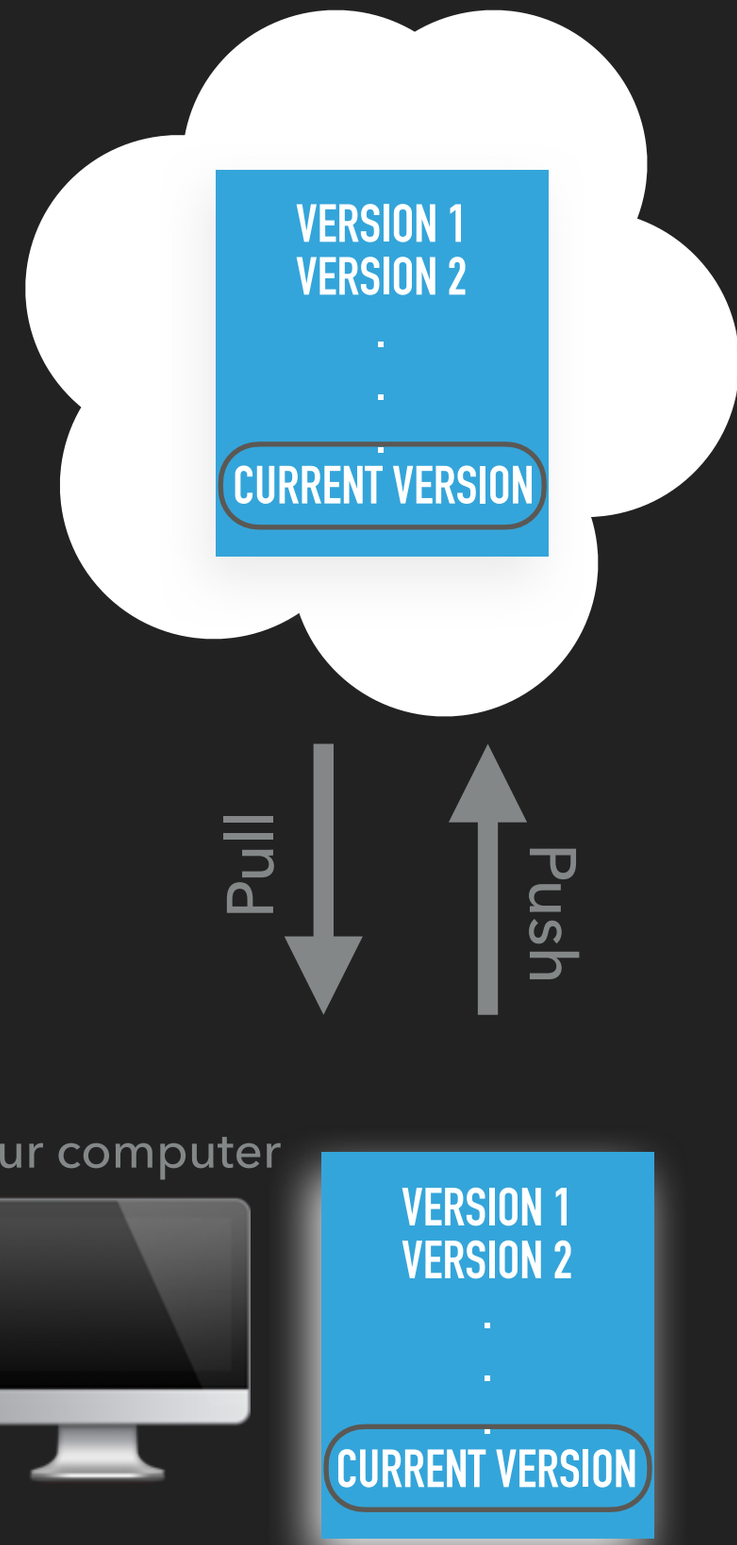
(make your edits)

```
$ git add <updated files>
```

```
$ git commit -m “your commit message”
```

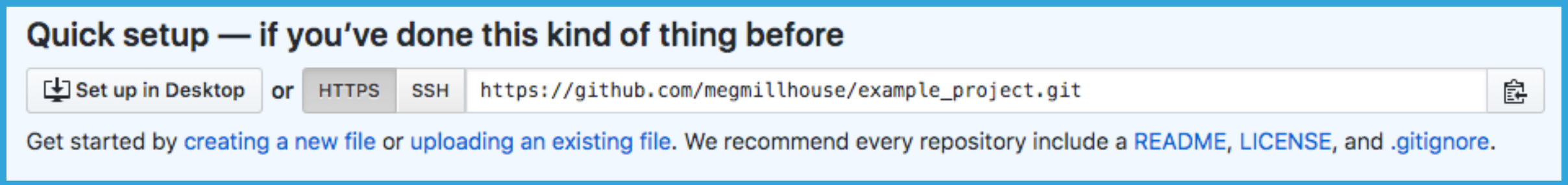
```
$ git pull
```

```
$ git push
```



# DO IT YOURSELF: SET UP A REMOTE REPOSITORY

https://github.com/new




```
$ git remote add origin <url>
```

```
$ git push --set-upstream origin master
```


```
$ git push --set-upstream origin master
```

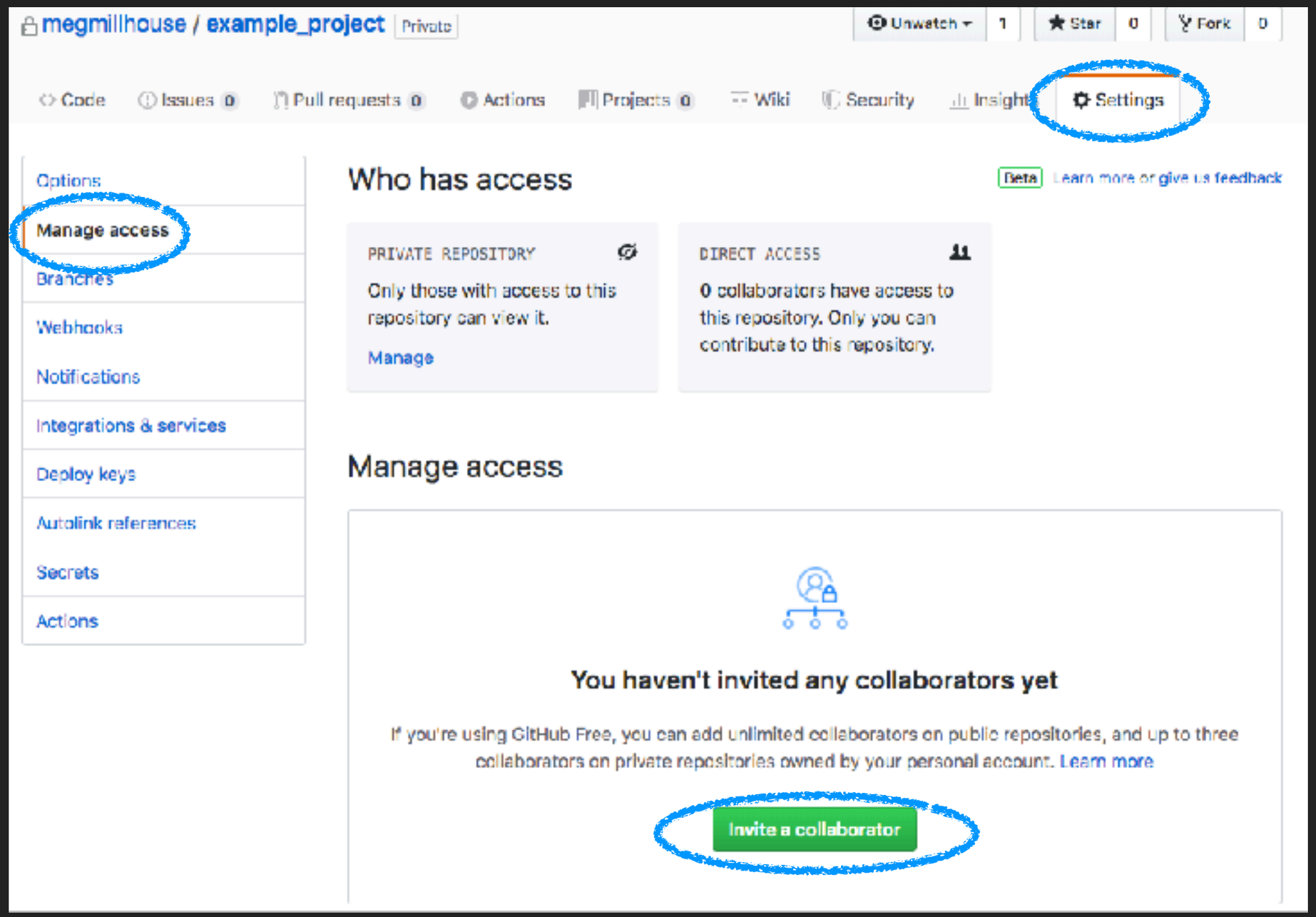
# COLLABORATION: SHARING CODE WITH OTHER PEOPLE

- You want to share your code with other people!
- You want to use or edit someone else's code!

What are the best ways to do that?

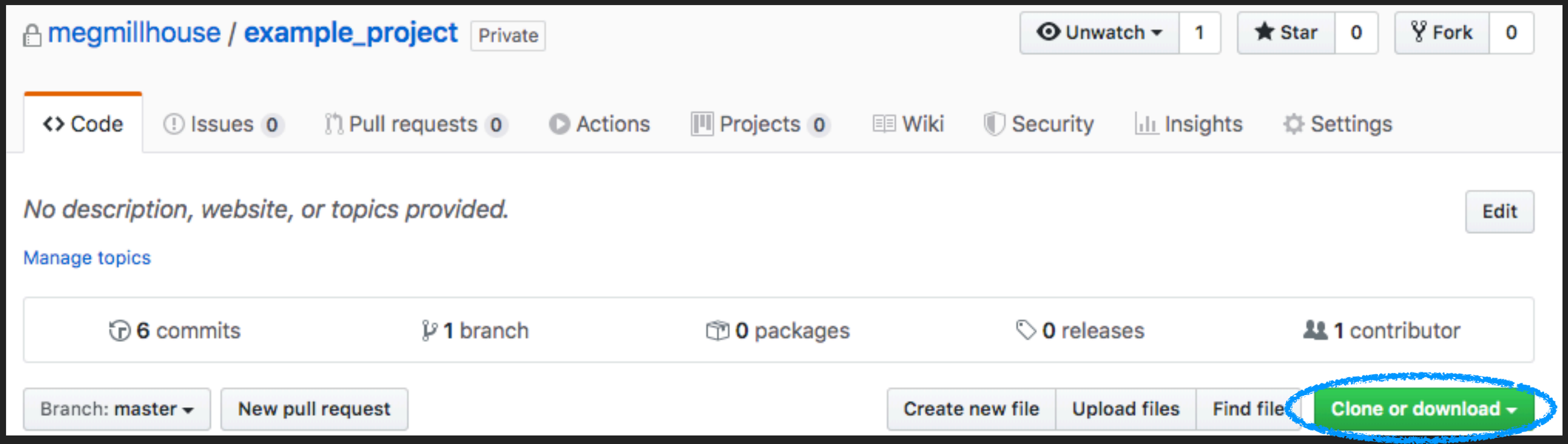
# COLLABORATION: SHARING CODE WITH OTHER PEOPLE

- Option 1: Invite collaborators to your project

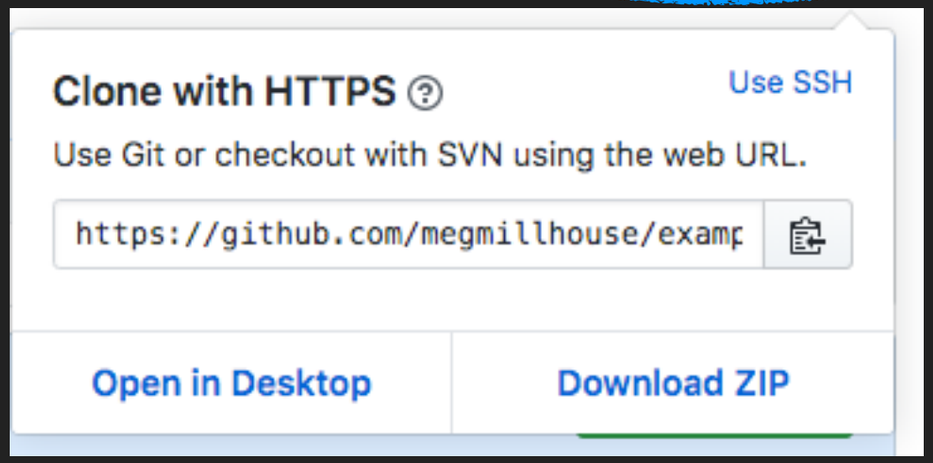


# COLLABORATION: SHARING CODE WITH OTHER PEOPLE

- Option 2: Clone an existing repository

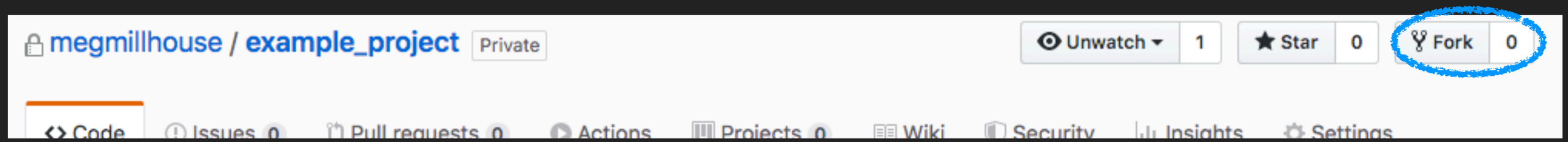



```
$ git clone <URL> <directory>
```



# COLLABORATION: SHARING CODE WITH OTHER PEOPLE

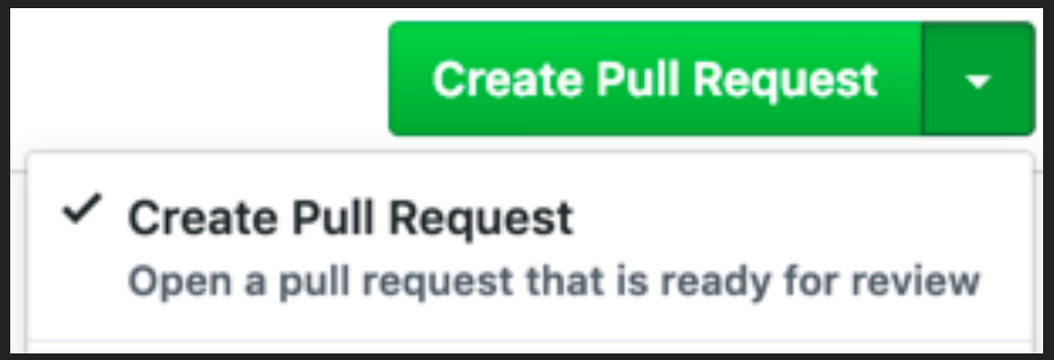
- Option 3: Fork an existing repository




- Forking a repository makes an independent copy
- Clone vs. Fork vs. Branch:
  - **Clone**: just want to use the code
  - **Fork**: Your own full version of the repository
  - **Branch**: temporary, more minor changes. Can have a branch within a fork

# FORKING

- One standard contribution workflow is the fork-branch-merge
  1. Fork a repository
  2. Make new branch in your fork for your particular update/feature
  3. Merge with master from a pull request



# GITHUB: WEB INTERFACE

- Lots of handy things you can do on the GUI

- (pull up GitHub here)

# LICENSING

- Licensing and best practices for use