Version Control (Lecture 8)

Peter Ganong, Maggie Shi, and Will Pennington

10/21/24

Table of contents I

Conceptual

Roadmap

▶ What is version control?
▶ Why do I need git?
▶ How (not) to learn git

What is version control?

▶ Version control: a system that records changes to a file or set of files over time so that you can recall specific versions later
▶ Examples of version control
  ▶ Informal: date multiple versions of a document
  ▶ Tools like dropbox and google docs do this automatically
  ▶ Another neat tool similar to google docs is Google Colab
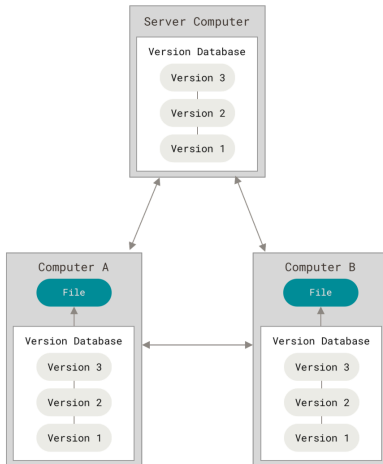▶ There is now even **VSCode for the web**... and it uses git for version control :-)

What is centralized version control?

▶ Centralized version control allows for easy collaboration across developers
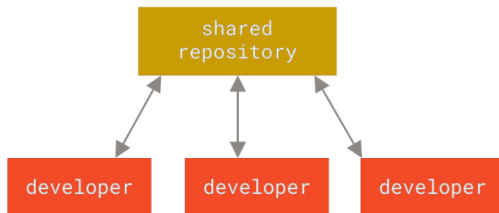
Git: "distributed" version control

▶ Each computer fully mirrors the repository, including its full history

How to learn git

Install git following the guide **here**

1. Accept that this is tricky and you will make mistakes
2. We are going to try a TON of different ways to teach this
   - graphical
   - code examples
   - do-pair-share in class
   - video game on pset
   - exercises on pset

Textbook

▶ We will be following **Pro Git**, by Scott Chacon and Ben Straub.

Section	Pro Git Chapter
Conceptual	Chapter 1 (Getting Started)
Track One Version on Local	Chapter 2.2 (Git Basics)
Branching, Merging	Chapter 3.2 (Git Branching)
Reconciling with Remote	Ch 2.5 and 3.5

note: this is a 500 page textbook! Barely scratching the surface. Pset material goes beyond lecture, but still only scratches the surface.

▶ Cheatsheet **here**

How not to learn git

▶ Command line git is what will be assessed on the quiz, the pset, and the final.
  ▶ Worst for learning: You might previously have used Github desktop or dragged and dropped files into github.com. These tools are easy to use, but the whole process is like magic and you never really understand what's happening.
  ▶ Also bad: Visual Studio Code's git panel is better than Github desktop and might ultimately speed you up, but don't use it right now. If you start using that then you won't understand as well what is happening under the hood.

Summary

- Git is a distributed version control system
- The only way we know to really understand it is via the command line

Track One Version on Local (Chapter 2.2)

Roadmap

- Git Project Sections
- Git workflow
- File lifecycle
- Basic Commands

Git Project Sections

1. Modified: you have changed a file, but not yet committed it to your database
2. Staged: you have marked a modified file go to go in your next commit
3. Committed: the file is stored in your local database
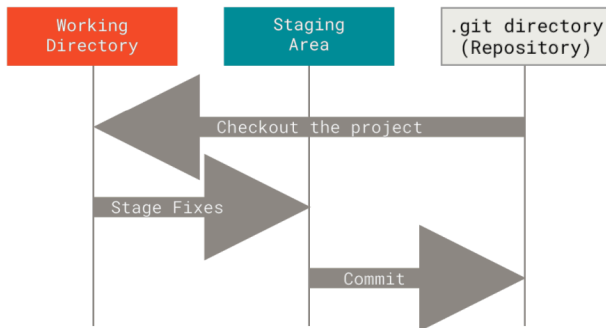
Remark: every git repo has a hidden `.git` folder containing the database

Git workflow

Git does not track every file that is in the directory containing
`.git`. Instead, you decide which files to track.

1. Modify files in your working tree
2. Selectively add files you want to commit to your staging area
3. Commit: permanently tracks the staged files as part of the
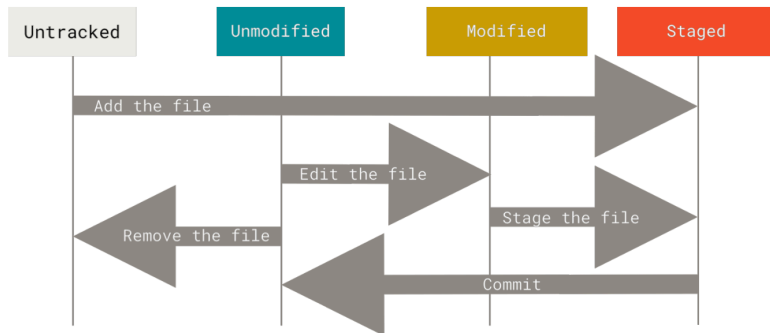   version control system

File lifecycle

Figure 8. The lifecycle of the status of your files

(Yes, it is a data visualization sin that they used the same colors from two slides ago to convey different concepts… very annoying)

Basic Commands
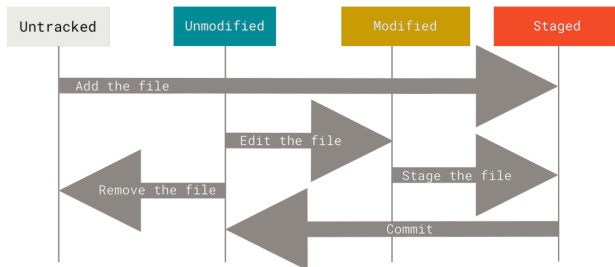
Figure 8. The lifecycle of the status of your files

`$ git status`
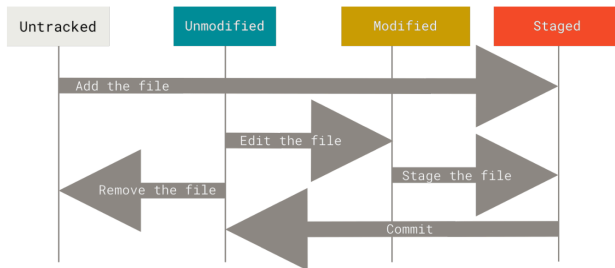
▶ check status of working directory

Basic Commands

Figure 8. The lifecycle of the status of your files

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

▶ add a single file to the staging area
▶ can be either the first arrow ("add the file") or, if the file is already tracked and was since modified, the third arrow ("stage the file")
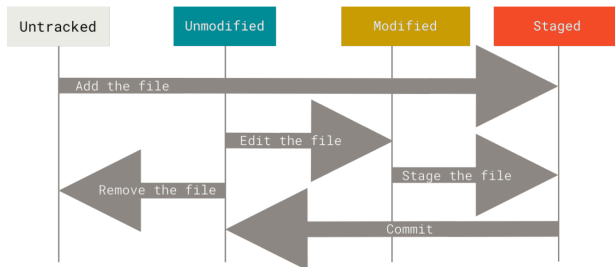
Basic Commands

Figure 8. The lifecycle of the status of your files

`$ git add .`

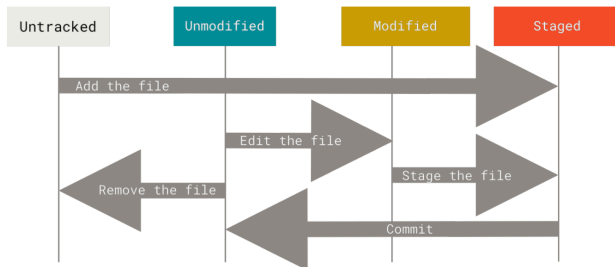▶ add all files (both untracked and modified) to the staging area

Basic Commands

Figure 8. The lifecycle of the status of your files

```
$ git commit -m "<message>"
```

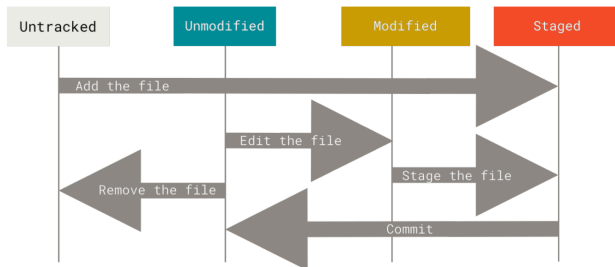▶ Commit files from staging area

Basic Commands

Figure 8. The lifecycle of the status of your files

`$ git commit -a -m "<message>"`

▶ Stage the already tracked files and then commit in one step

Create a new repo

https://github.com/new and then clone to your computer

Choices

1. Public or private?
   - ▶ Obviously anything with sensitive data or code should be private
   - ▶ Otherwise, we think it's better to leave stuff public. Public repos make it easier for a potential employer to evaluate and be confident in your coding ability.
2. License – Ganong's lab uses the MIT license. Key thing is to specify **a** license
3. `.gitignore` – next slide

Git Ignore I

▶ You will often have files or filetypes that you want Git to systematically avoid (*ignore*). Examples:
  ▶ Automatically-generated files from compiling Python (`.pyc`)
  ▶ Large data files (this will become very important on PS4)
▶ Create a file called `.gitignore` to tell git which files to ignore
▶ You must commit your `.gitignore` to the repo

Git Ignore II

Example .gitignore :

```
# ignore all .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in any directory named build
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories
doc/**/*.pdf
```

▶ See more examples **here**
▶ Github's **recommendation** for Python. We haven't tested it yet.

Summary

▶ Adding files to the staging area prepares Git to track changes
▶ Committing changes allows Git to take a snapshot of your files
▶ Use `.gitignore` to ignore irrelevant files

Track Multiple Versions on Local – Branching
(Chapter 3)

Roadmap

- Branching: overview
- Creating a New Branch
- Switching Branches
- Other Useful Branching Commands
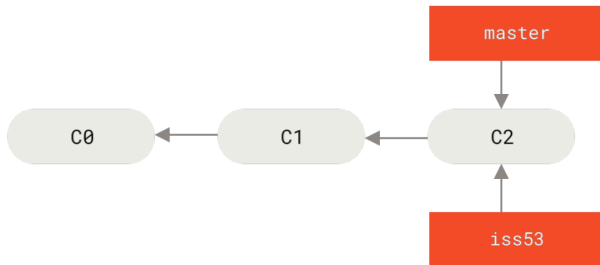
Branching

▶ A simple commit history

Remarks:

▶ These commit names (`C0`, `C1`…) are terrible. Insufficiently descriptive. Using them only because we are following an example from the textbook.
▶ Git today calls the primary branch `main`. However, git used to call the primary branch `master`. The textbook uses this legacy terminology.

Branching
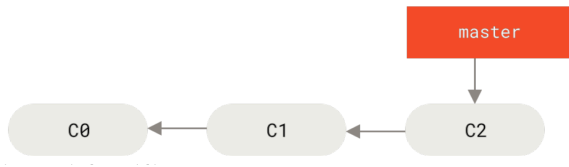
▶ What would you do if you wanted to do some exploratory work in a safe environment without affecting the work other team members might be doing?

Creating a New Branch


```
$ git branch iss53
```

▶ Creates a new branch called iss53

Switching Branches I
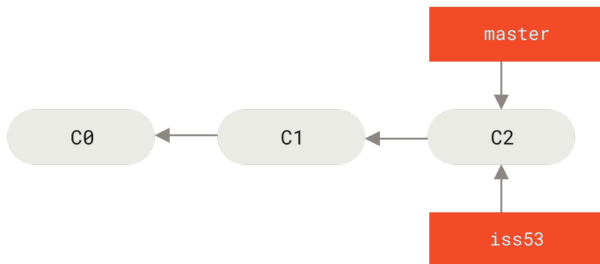

```
$ git switch iss53
```

▶ Switches to your new branch iss53

Switching Branches II
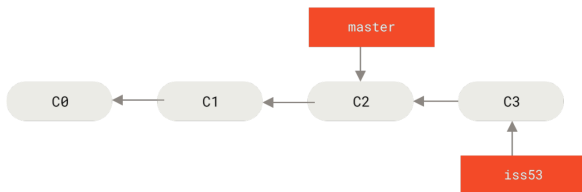

```
$ git commit -m "New work on iss53"
```

▶ With `iss53` checked out, your commits now move `iss53`
  forward while leaving `master` untouched

Other Useful Branching Commands

```
$ git branch
```

▶ Returns a listing of your current branches

```
$ git branch -d <branch_to_delete>
```

▶ Deletes *branch_to_delete*

Summary

- Working in branches creates an independent, safe development environment
- Commits only modify current branch, leaving others untouched

Reconcile Multiple Versions on Local – Merging (Chapter 3)

Roadmap

- Fast forward merge
- Three way merge
- Three way merge with a conflict
- Do-pair-share
- Resolving file-level conflicts

Fast forward merge I
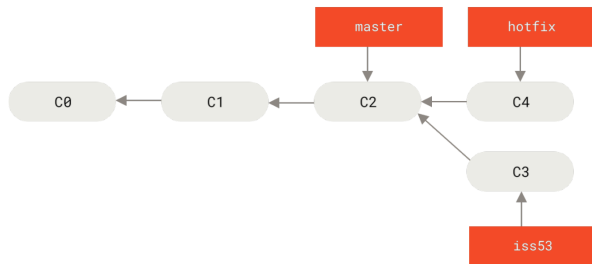

```
$ git switch master
$ git branch hotfix
$ git switch hotfix
$ git commit -a -m "Work in hotfix branch"
```

▶ Suppose now that you want to switch back to master and make adjustments in a new hotfix branch

Fast forward merge II
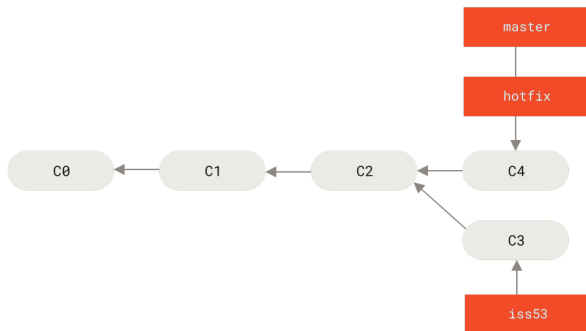

```
$ git switch master
$ git merge hotfix
```

▶ Brings changes from `hotfix` into `master`
▶ This is a called a "fast forward merge"

Three way merge I
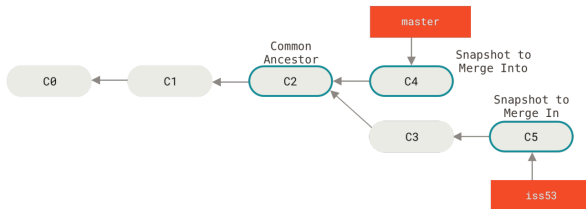
▶ Merging `iss53` into `master` now has to account for a divergent work history

Three way merge II
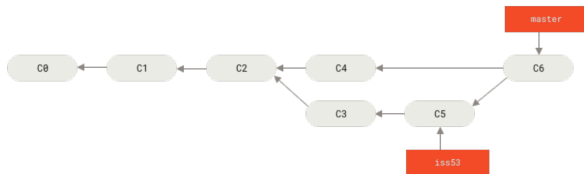

```
$ git switch master
$ git merge iss53
```

▶ This is called a "three way merge" because it involves
   combining C2, C4, and C5 into a single unified version of the
   code

Three way merge with a conflict I

We will use the repo **merge_example** which has the following commit history:

name	branch	content	parent	conflict?
C0	`main`	Create `add.py`, which adds x to 3	none	no
C1	`hotfix`	Change `add.py` to add x and y	C0	no
C2	`iss53`	Change `add.py` to add a list	C1	no
C3	`main`	Change `add.py` to add x to 4	C2	no
C4	`main`	Merge `hotfix` into `main`	C1 & C3	yes!

Three way merge with a conflict II

▶ Commit history visualization
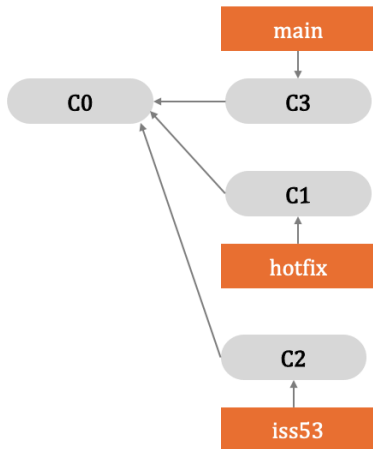
Three way merge with a conflict III

▶ We will try to merge hotfix into main
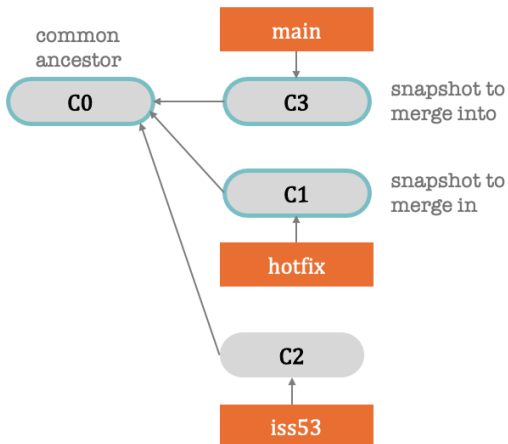
Three way merge with a conflict IV

```
'''
A function that adds x to 4
'''
# define function
def add(x):
# arguments: x (number to add to 4)
    return (x + 4)
```

$ git switch main

```
'''
A function that adds a list of numbe
'''
# define function
def add(x):
# arguments: x (list of numbers to b
    return sum(x)
```

$ git switch iss53

```
'''
A function that adds x to y
'''
# define function
def add(x, y):
# arguments: x, y (numbers to add)
    return (x + y)
```

$ git switch hotfix

▶ Running *git switch* command switches to different branches,
  and each branch contains a different version of *add.py*

Three way merge with a conflict V

```
$ git switch main
$ git merge hotfix
```


▶ Merging *hotfix* into main causes conflicts.

Three way merge with a conflict V

```
$ git switch main
$ git merge hotfix
```


▶ Merging *hotfix* into main causes conflicts.

▶ Resolve within Visual Studio Code.
   ▶ Remark: Although merge conflicts can be resolved via command line, it is more time consuming and provides no educational benefit to do so.

▶ Discussion question: Do we want "Current Change", "Incoming Change" or "Both"? Why?

Three way merge with a conflict VI

How add.py looks after conflict is resolved * Resulting file in main looks exactly like version in hotfix

```
'''
A function that adds x to y
'''
# define function
def add(x, y):
# arguments: x, y (numbers to add)
    return (x + y)
```

Three way merge with a conflict VII

:::
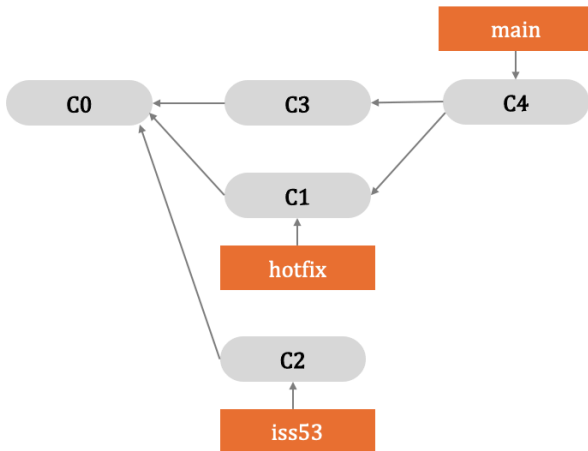
do-pair-share: merge conflicts
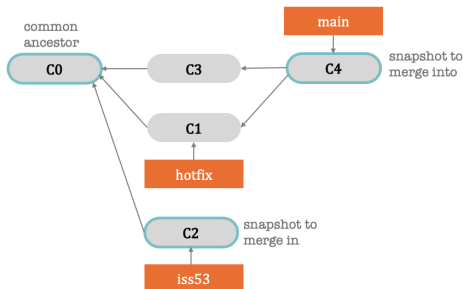
Goal: the file *add.py* should look identical on `main` and `iss53`

1. `git fork`
   `https://github.com/uchicago-harris-dap/merge_example`
2. Clone your fork to local
3. Merge `iss53` into `main`, naming your commit C5
4. Using VSCode, resolve the conflict in favor of the `iss53` version

do-pair-share: Solution

File versions:

```
'''
A function that adds x to y
'''
# define function
def add(x, y):
# arguments: x, y (numbers to add)
    return (x + y)

$ git switch main
```
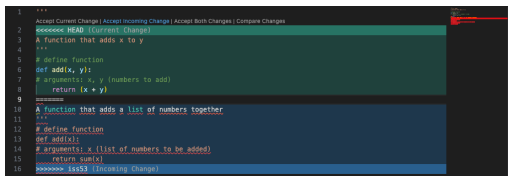
```
'''
A function that adds a list of numbers together
'''
# define function
def add(x):
# arguments: x (list of numbers to be added)
    return sum(x)

$ git switch iss53
```

do-pair-share: Solution

```
$ git switch main
$ git merge iss53
```

▶ Following *git merge* command, resolve conflict by selecting "Accept Incoming Change" in Visual Studio Code

do-pair-share: Solution

How `main` looks after conflict is resolved

```
'''
A function that adds a list of numbers together
'''
# define function
def add(x):
# arguments: x (list of numbers to be added)
    return sum(x)
```

Resolving file-level conflicts I

▶ For some file types, line-by-line adjustments will not be possible
  ▶ E.g. PDFs

Resolving file-level conflicts II

Repo

▶ Branch *iss1* has a version of the file *Example.pdf* that conflicts with the version on *main*
▶ How to resolve the conflict when merging *main* into *iss1*?
▶ Pull request **here**

Resolving file-level conflicts III

```
$ git switch iss1
$ git merge main
```

▶ After merge step, git returns a conflict message.

Resolving file-level conflicts III

```
$ git switch iss1
$ git merge main
$ git checkout --theirs Example.pdf
$ git add Example.pdf
$ git commit -m "C5: Merging main into iss1"
```

▶ Solution: *checkout* command with
  ▶ –theirs: keep file from incoming branch (main)
  ▶ –ours: keep file from current branch (iss1)
▶ Add changes and commit to finalize merge

Summary

▶ Merging brings changes from one branch into another
   ▶ Easier case: fast forward
   ▶ Harder case: three way merge
▶ When branches have divergent commit histories, you may have to manually resolve conflicts
▶ Some conflicts must be resolved by keeping files from one branch or another

Reconciling Your Version with a Remote
(Chapter 2.5 and 3.5)

Roadmap

- ▶ list all branches
- ▶ fetch
- ▶ pull
- ▶ pull requests
- ▶ push

see all branches

▶ list all branches, including branches in the remote repository which are not on your computer

```
$ git branch -a
```

fetch

```
$ git fetch
```

▶ Get changes from remote repository to your computer.
▶ This command does not modify any of your existing work.
▶ If you did some work in the same branch, you will need to do a *git merge origin* to reconcile your work with the work in the remote.

pull

▶ *git pull* equivalent to *git fetch* followed by *git merge*

```
$ git pull   <branch_name>
```

▶ Automatically fetches branch from the remote repository, then merges that branch into your current branch
▶ If there are reconcilable changes in both places, git will create a reconciling merge commit and ask you to confirm the content of the git message.
▶ If there are irreconcilable changes, then you are in the world of the last section of lecture where you need to do some reconciliation.

Pull Requests

▶ A pull request is a way to merge changes from a branch into an upstream repository.

▶ Doesn't `git merge` already do this? Yes, but it this way allows for collaborative peer review

▶ Example: open pull request for merge_example repo (here)

Pull Requests

▶ Pull requests allow for line-by-line comparison of changed files
  ▶ Click "+" to add a comment (show in browser)
  ▶ They also identify merge conflicts

git merge vs pull requests

When should I use each?

▶ Use pull requests when you want to preview a change
▶ Use pull requests when you want someone else to review the change
▶ Otherwise, just use `git merge`

push

```
$ git push  <branch_name>
```

- push (committed) changes from local to remote repository
- what happens if you push and there are also changes in the remote? "Updates were rejected because the remote contains work that you do not have locally. This is usually caused by another repository pushing to the same ref. You may want to first merge the remote changes (e.g., hint: 'git pull') before pushing again."

Summary

- *fetch* brings changes from remote to local
- *pull* brings changes from remote to local and tries to reconcile via merge commit
- pull requests provide a "trial run" of merges that is visible to collaborators
- *push* sends changes from local to remote
- Remark: "sync changes" (appears in GH desktop and VSCode) really means push and pull