Version Control (Lecture 8)

Peter Ganong, Maggie Shi, and Will Pennington

2024-10-21

Table of contents I

Conceptual

Roadmap

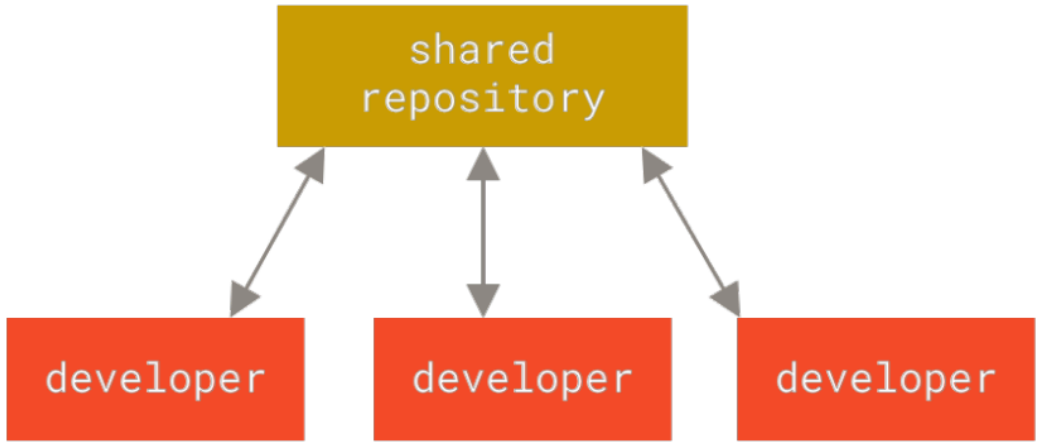
- What is version control?
- What is Git and why do I need it?
- How (not) to learn git
- Git vs. Github vs. Github Desktop

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 :-)
- Cloud tools not a good version control solution for code because *code needs to run*

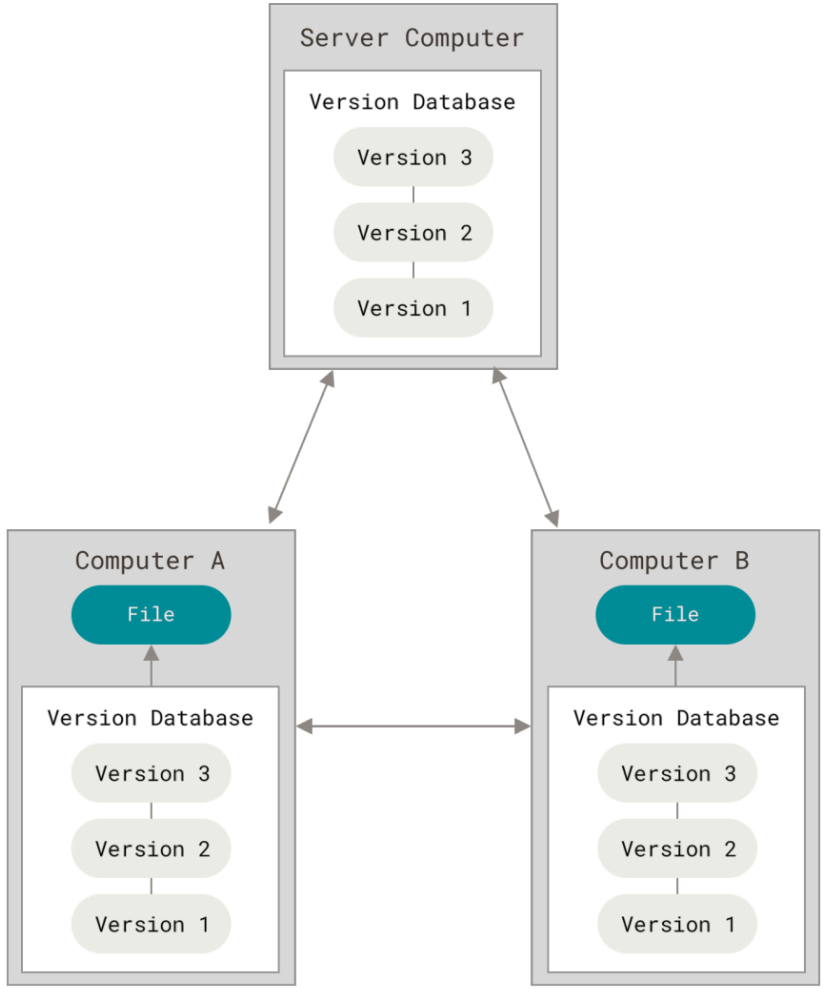
What is centralized version control?

- Centralized version control allows for easy collaboration across developers



Git: “distributed” version control

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



Why Git?

- We will cover Git for the second half of today and next lecture. You might ask: what's in it for me?
- In general:
  - Part of the standard toolbox for analysts, but as you will see, it is not an easy skill to pick up on the job
  - Useful for solo projects for version control (as opposed to: `final_report_revised_v5_maggieedits.qmd`)
  - Crucial in group projects to make sure you don't write over others' work or break something.

Why Git?

- In this class:
  - Use command line git to submit PS3, 4, 5, 6 and final project
  - Collaboration in PS3, 4, and final project – can work parallel as opposed to sequentially, and work without fear of writing over others' work
  - Resolve merge conflicts related to changes in the student repo

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

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.

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 (link)**

Git vs. Github vs. Github Desktop

- Git: software used for version control locally (i.e., on your computer)
  - The "object" is a git repository
  - Was installed automatically when you downloaded Github Desktop
- Github: hosting service for git repositories – i.e., a place to store them online
  - There is one version of your repo online and another that is local, and you manually choose when the sync them ("push" and "pull")
- Github Desktop: software to interact with Github
  - Introduced in 2017: "industry veterans" will not know what this is
  - Under the hood, it is doing command line git

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 workflow
- File lifecycle
- Basic Commands

Git workflow

Every git repo has a hidden `.git` folder containing the database

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

Analogy: in the standard workflow, steps 2 and 3 happen automatically after you save. In Git, you choose when this happens to ensure you don't "save" something that breaks the code.

File lifecycle

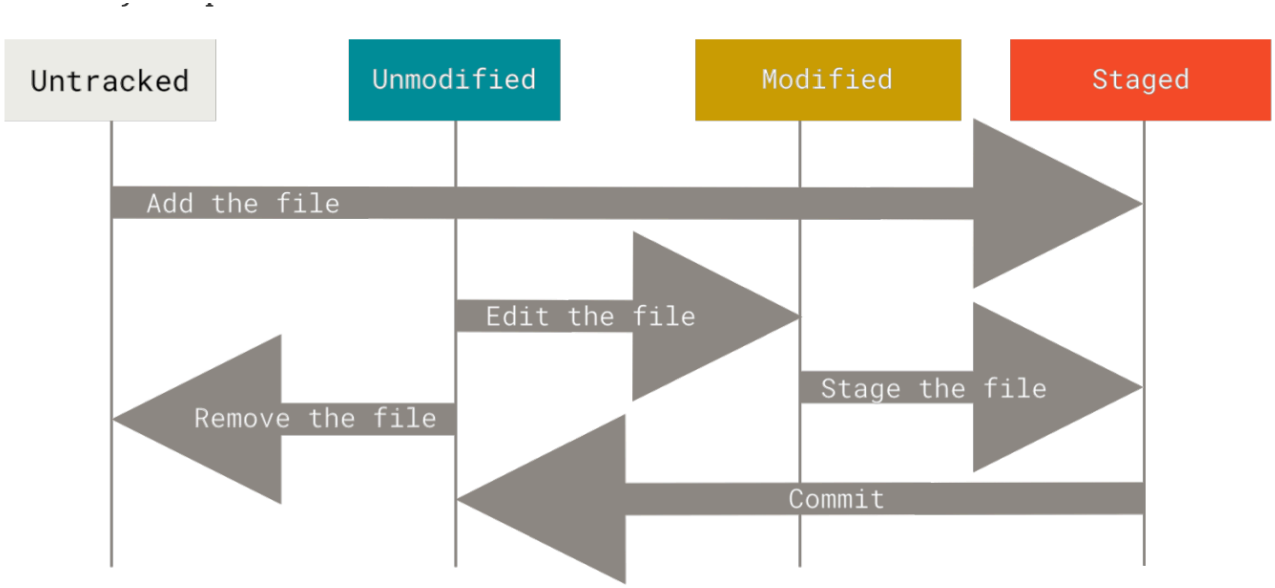


Figure 8. The lifecycle of the status of your files

Basic Commands

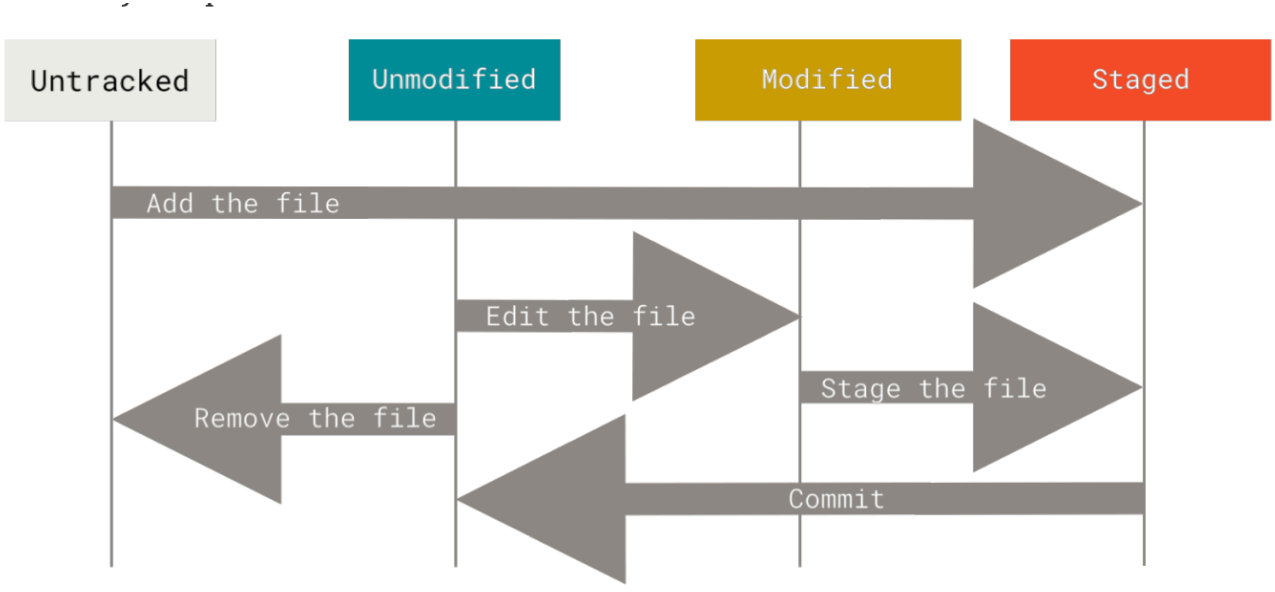


Figure 8. The lifecycle of the status of your files

```
$ git status
```

- check status of working directory: what's staged for commit, what's tracked but not staged, and what's not tracked (but modified)

Basic Commands

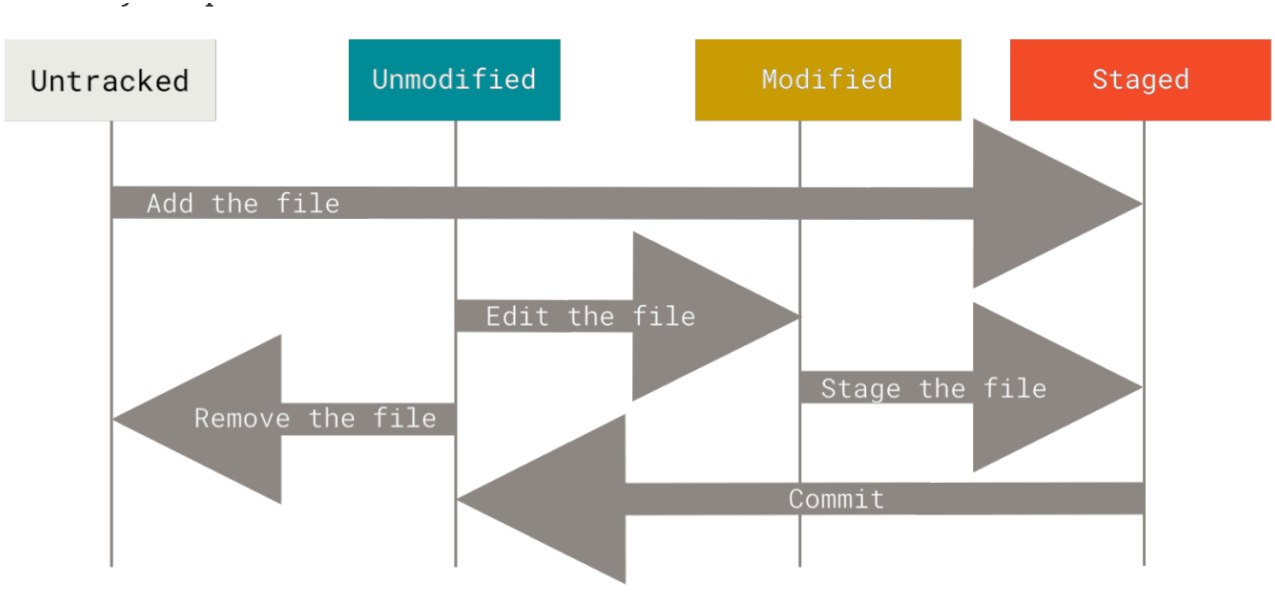


Figure 8. The lifecycle of the status of your files

```
$ 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

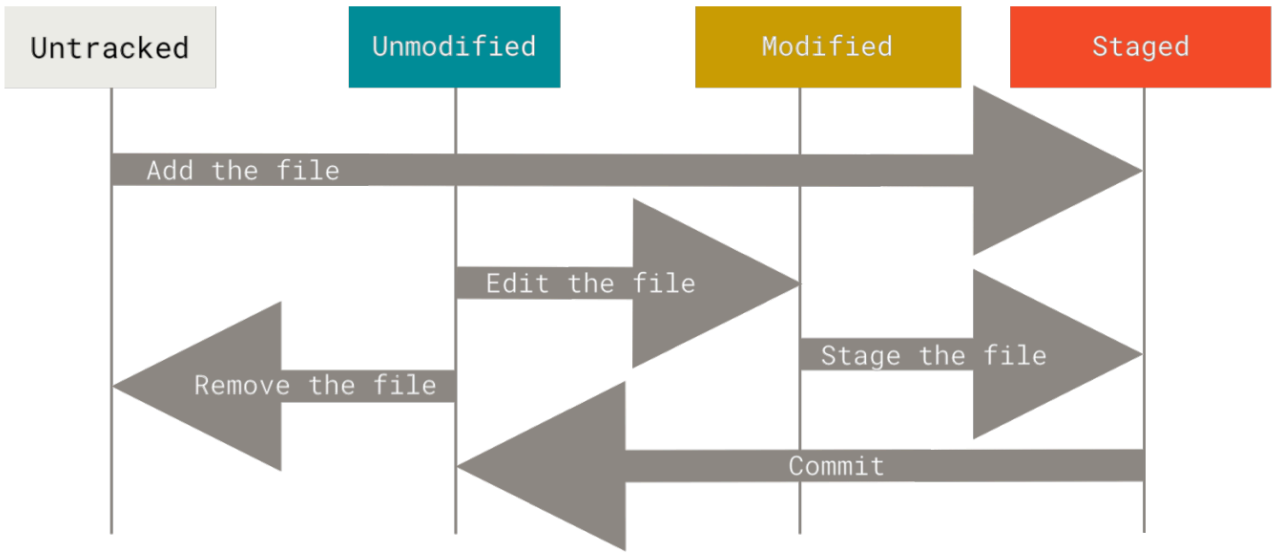


Figure 8. The lifecycle of the status of your files

```
$ git add .
```

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

Basic Commands

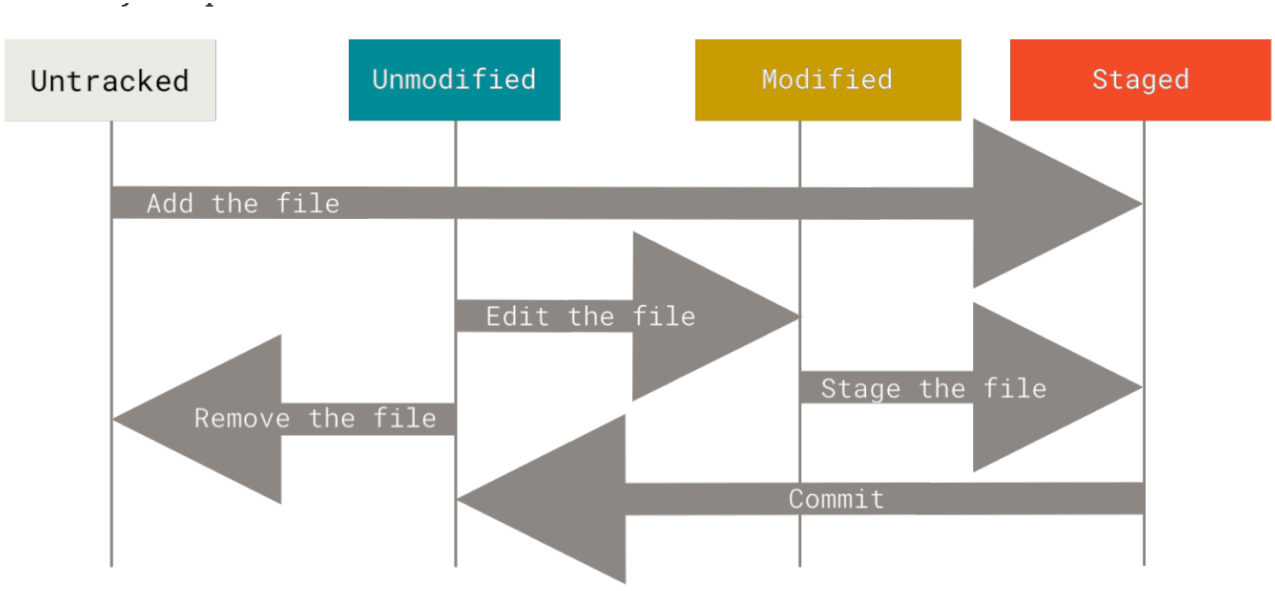


Figure 8. The lifecycle of the status of your files

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

- Commit files from staging area with a message.
- The message should be informative about what's in the commit: "debugged function to do XYZ"

Basic Commands

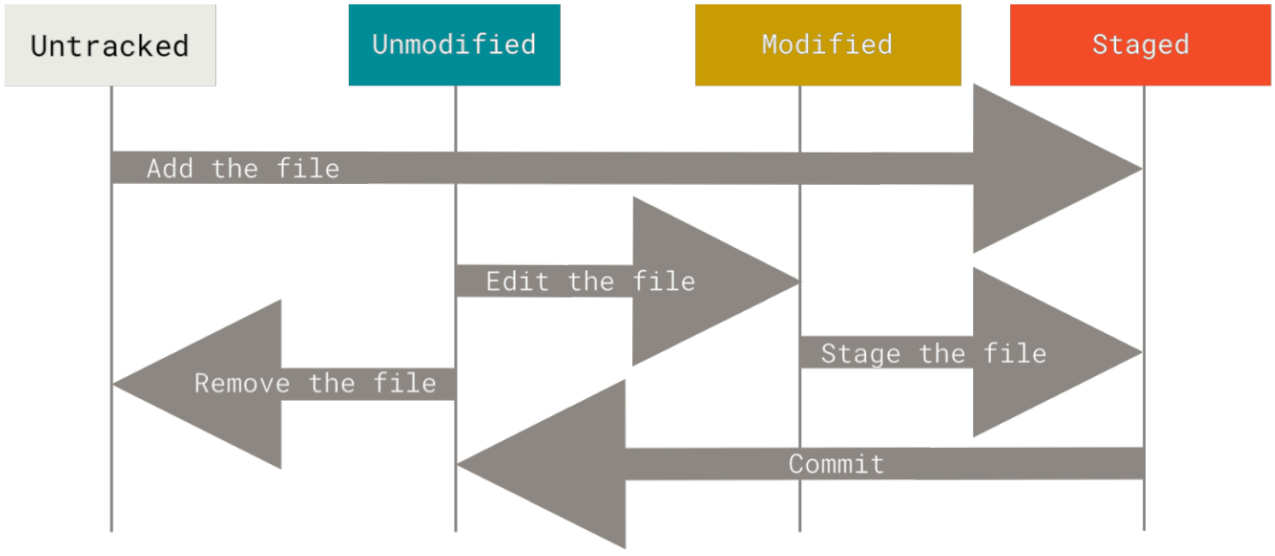


Figure 8. The lifecycle of the status of your files

```
$ git log
```

- prints log of recent commits

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)
  - Mac users: `.DS_Store`!
- 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 (link)**
- Github's **recommendation (link)** 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

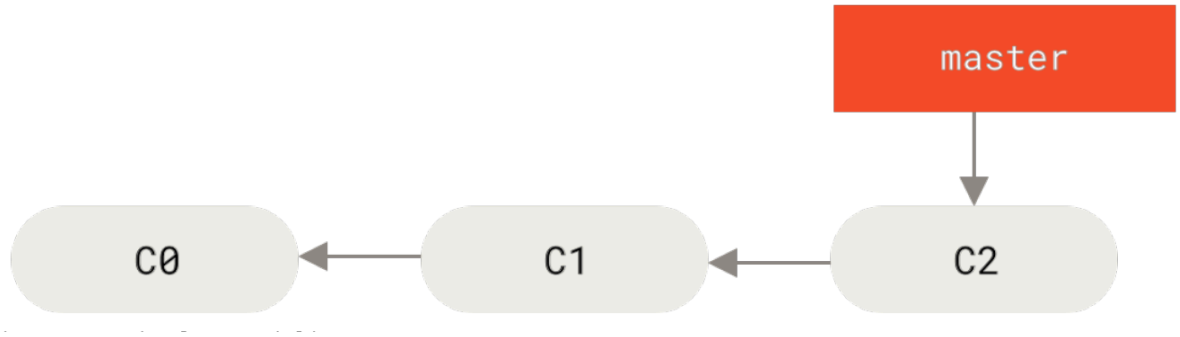
Track Multiple Versions on Local – Branching (Chapter 3)

Roadmap

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

Branching

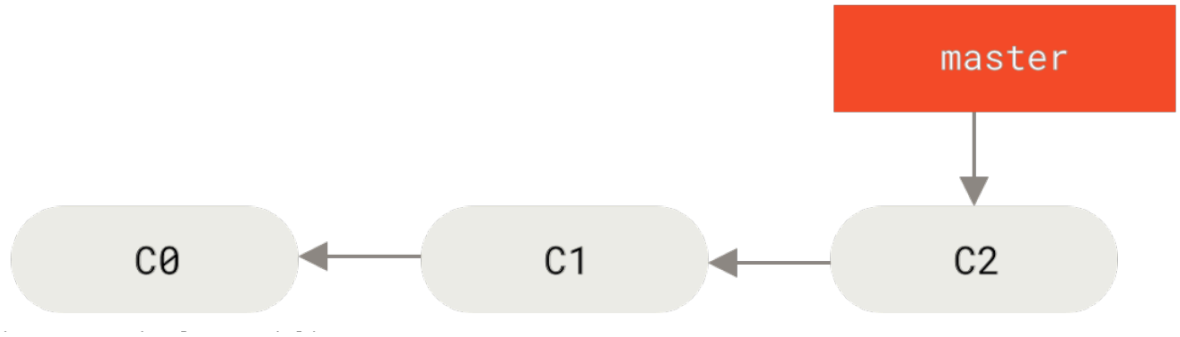
- Let's say our git repo has the file `mycode.py` and the following commit history



- Some terminology: `HEAD` is your current position in the Git repo's history
- Right now, the `HEAD` is in the `master` branch at commit `C2`

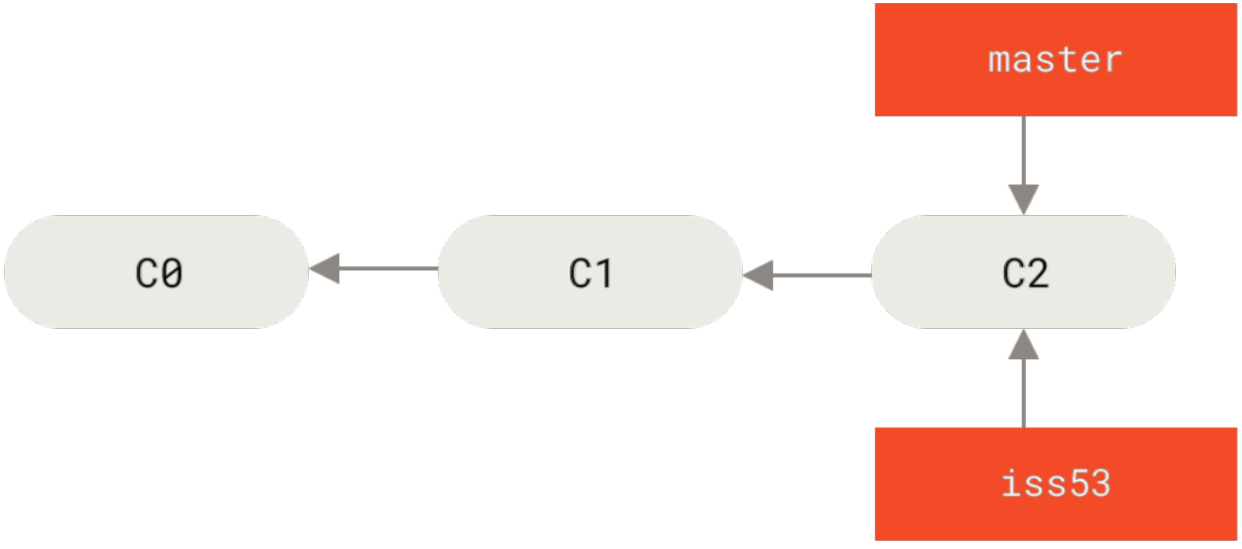
Branching

- Let's say our git repo has the file `mycode.py` and the following commit history



- Some terminology: `HEAD` is your current position in the Git repo's history
- Right now, the `HEAD` is in the `master` branch at commit `C2`
- Say we want to do some exploratory work in a safe environment without affecting the work other team members are doing –> *create a new branch*

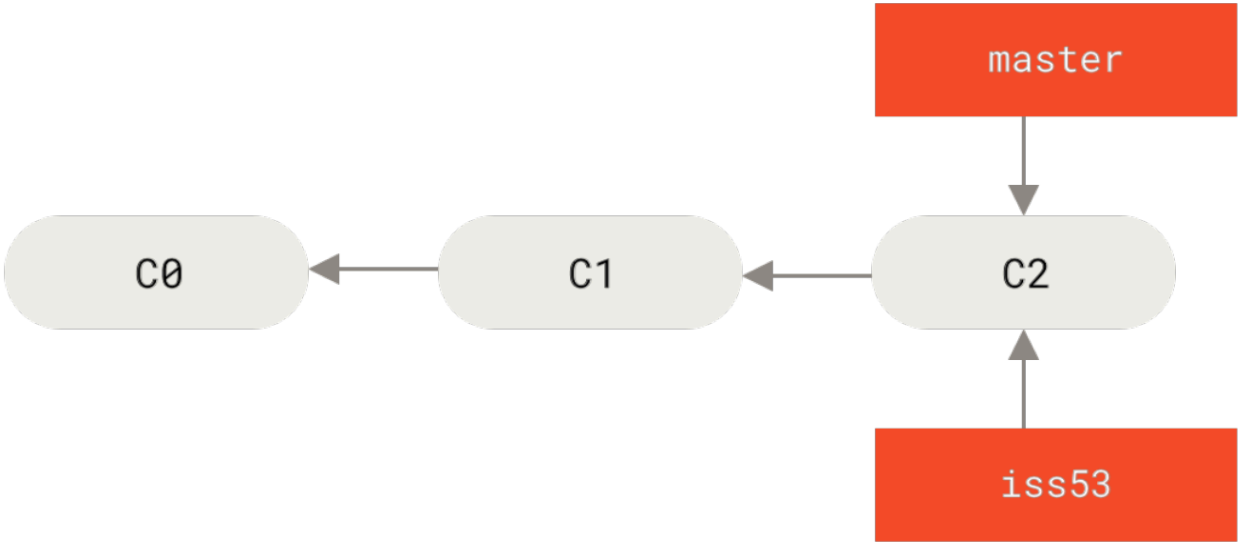
Creating a New Branch




```
$ git branch iss53
```

- Let's call our new branch `iss53` (stands for issue 53)

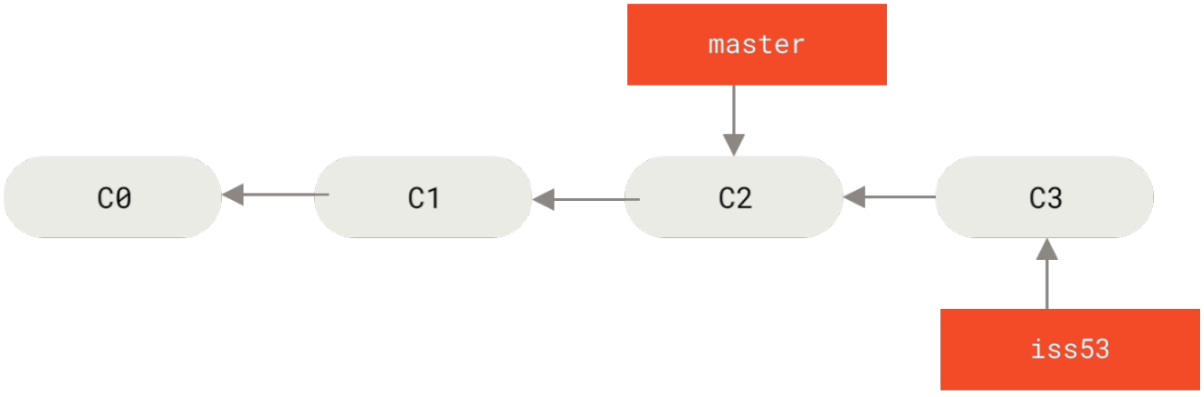
Switching Branches I




```
$ git switch iss53
```

- Switches our `HEAD` (i.e., where we are) to our new branch `iss53`

Switching Branches II




```
[make some changes to file.py]
$ git commit -m "commit message for C3"
```

- With `iss53` checked out, our commits now move `iss53` along with `HEAD` *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* – you only want to do this after you've merged in your work to another branch (up next)

Summary

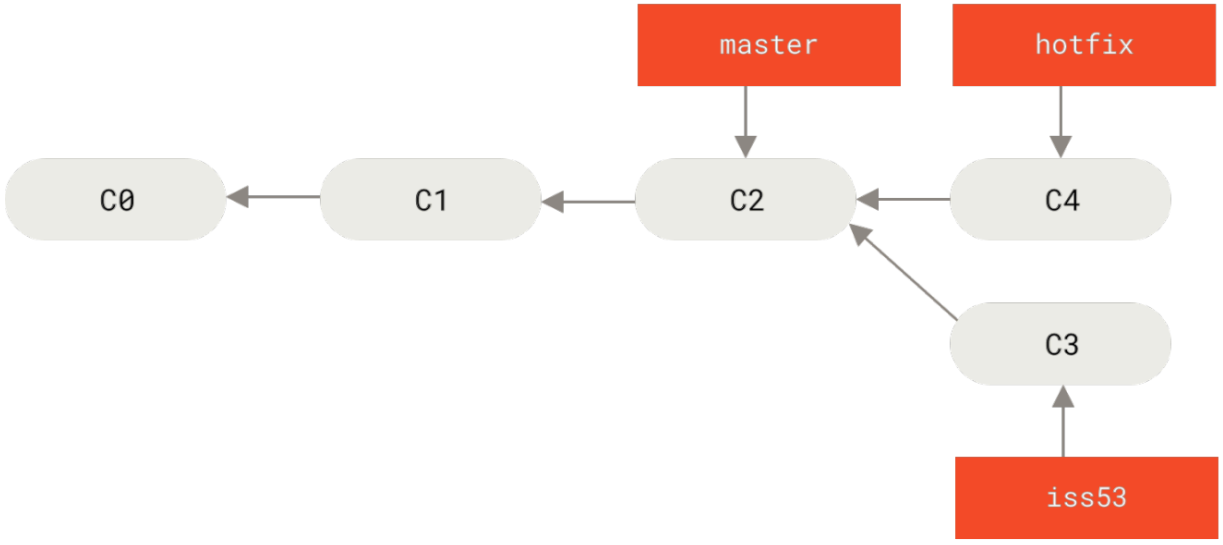
- Working in branches creates an independent, safe development environment
- Commits only modify current branch, leaving others untouched

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

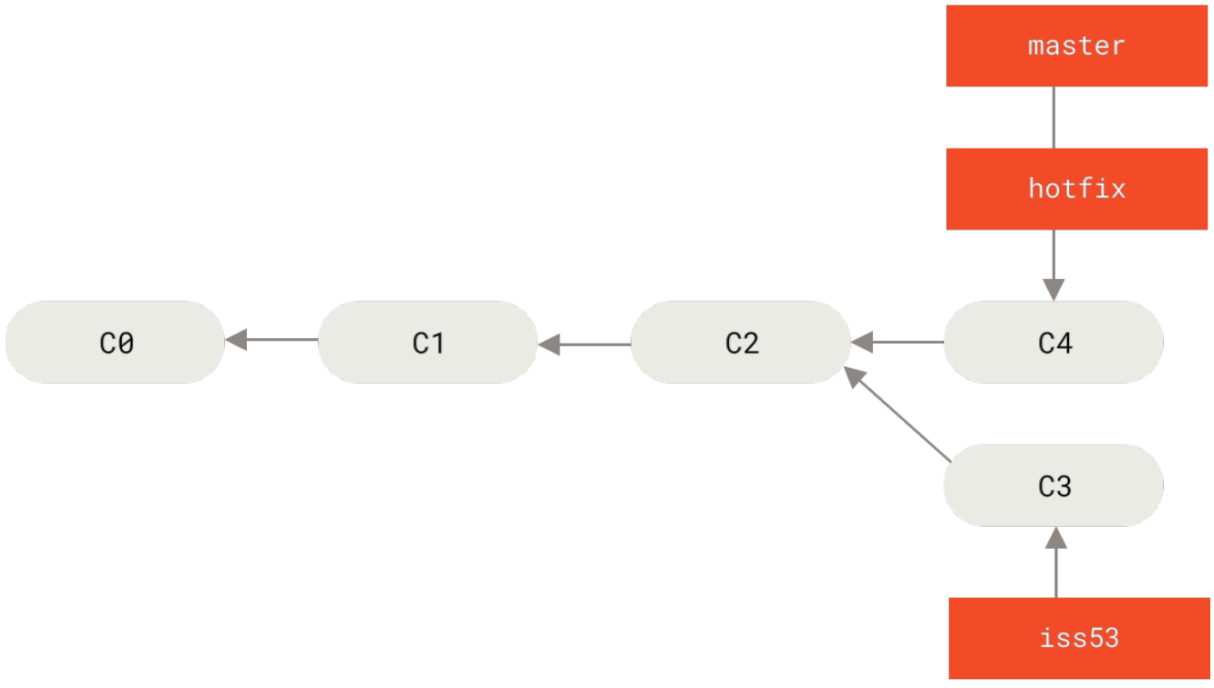
Fast forward merge I




```
$ git switch master
$ git branch hotfix
$ git switch hotfix
[make some changes to myfile.py]
$ git add myfile.py
$ git commit -m "commit message for C4"
```

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

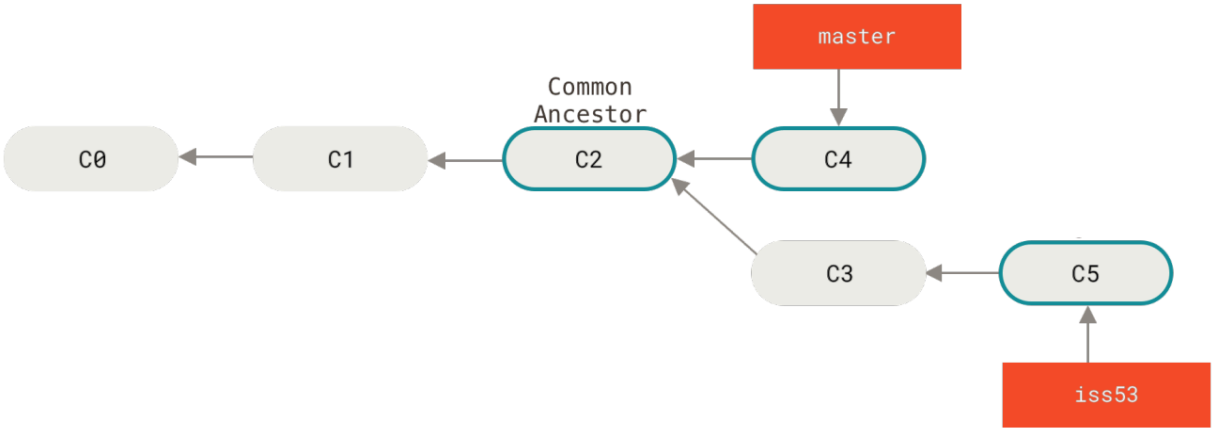
Fast forward merge II




```
$ git switch master
$ git merge hotfix
$ git branch -d hotfix
```

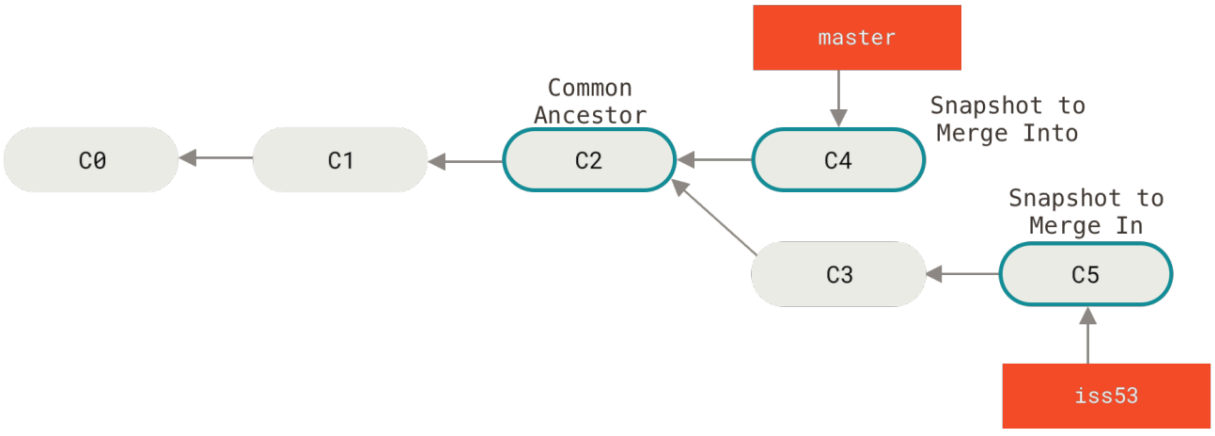
- Brings changes from `hotfix` into `master`: *"fast forward merge"*
- You can now safely delete the branch `hotfix`

Three way merge I



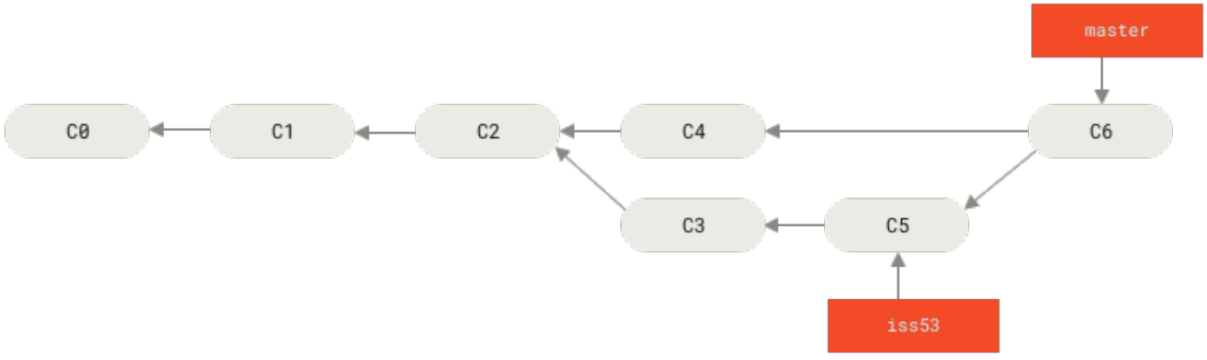
- Now say that as you were working on merging `hotfix` into `master`, your partner was working on `issue53`
- And they also made some changes to `myfile.py`, which they committed as "C5"

Three way merge II



- Merging `iss53` into `master` now has to account for a divergent work history

Three way merge III




```
$ 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
- Because you might have to change C4 or C5 in order to get them to merge, this creates a new commit called C6

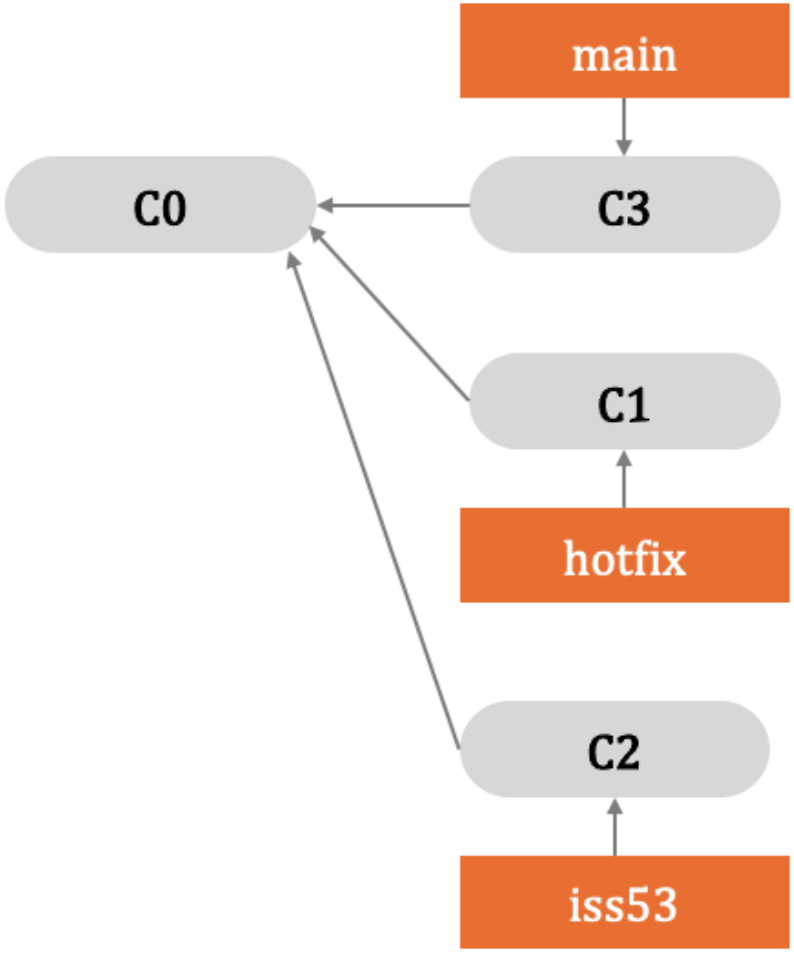
Three way merge with a conflict I

- Let's now switch examples and re-build our toy
- 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	C0	no
C3	`main`	Change `add.py` to add x to 4	C0	no

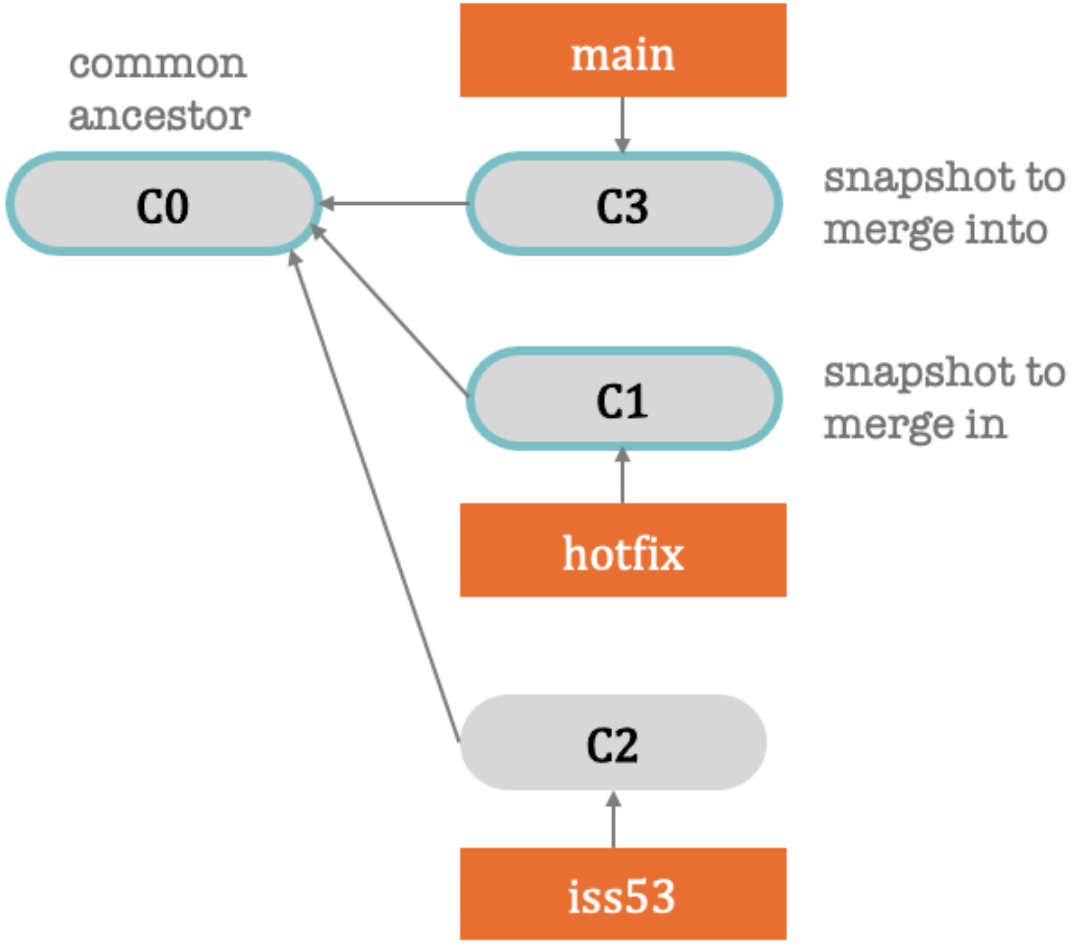
Three way merge with a conflict II

- 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

- Now let's try to merge `main` and `hotfix` together
- Doing so will create a new commit, C4

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	C0	no
C3	`main`	Change `add.py` to add x to 4	C0	no
C4	`main`	Merge `hotfix` into `main`	C1 & C3	yes!

Three way merge with a conflict VI

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

Three way merge with a conflict VI

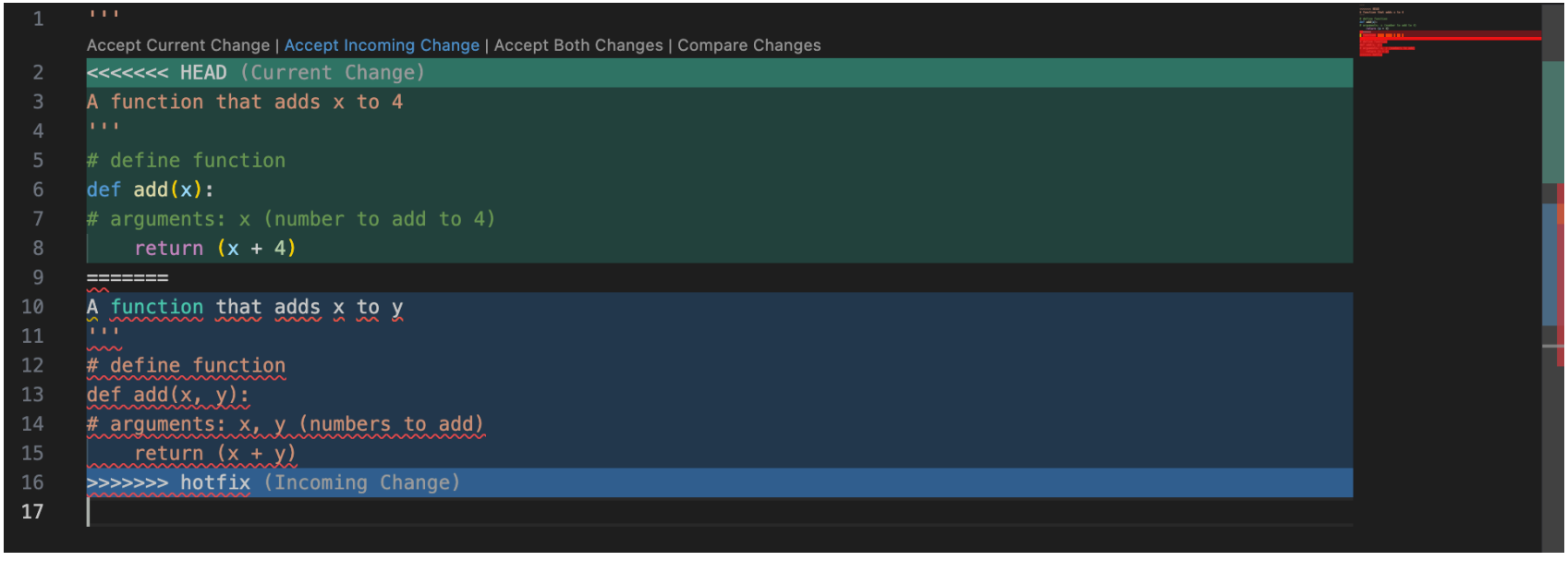
```
$ git switch main
$ git merge hotfix
```

- Because we have different versions of `add.py` in each branch, merging `hotfix` into `main` causes conflicts!
- You will get this scary-seeming message – *don't panic!*

```
Auto-merging add.py
CONFLICT (content): Merge conflict in add.py
Automatic merge failed; fix conflicts and then commit the r
```

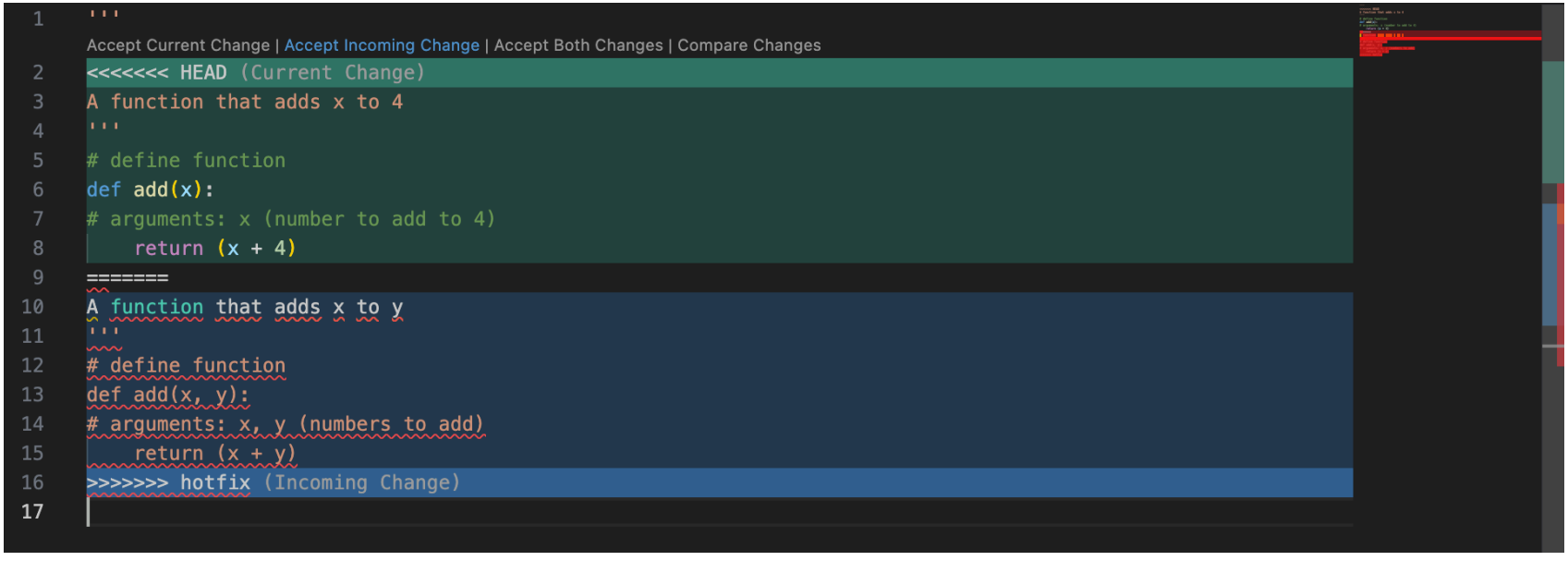
Three way merge with a conflict VII

- After you type `git merge hotfix`, git will edit your `add.py` file and use the following format wherever it finds a conflict:



Three way merge with a conflict VII

- After you type `git merge hotfix`, git will edit your `add.py` file and use the following format wherever it finds a conflict:



- Resolve within Visual Studio Code by clicking “Accept Current Change” or “Accept Incoming change” or “Accept Both changes”

Three way merge with a conflict VI

- In this case, we'll keep what's in `hotfix`
- How `add.py` looks after we've clicked "Accept incoming change": 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)
```

- (Of course, you could have resolved it by "Accepting current change", which would look exactly like `main`!)

Three way merge with a conflict VII

- After resolving, have to add and then commit the merge

```
git add add.py
git commit -m "C3: resolved conflict between main and hotfi
```

Three way merge with a conflict VII

- After resolving, have to add and then commit the merge

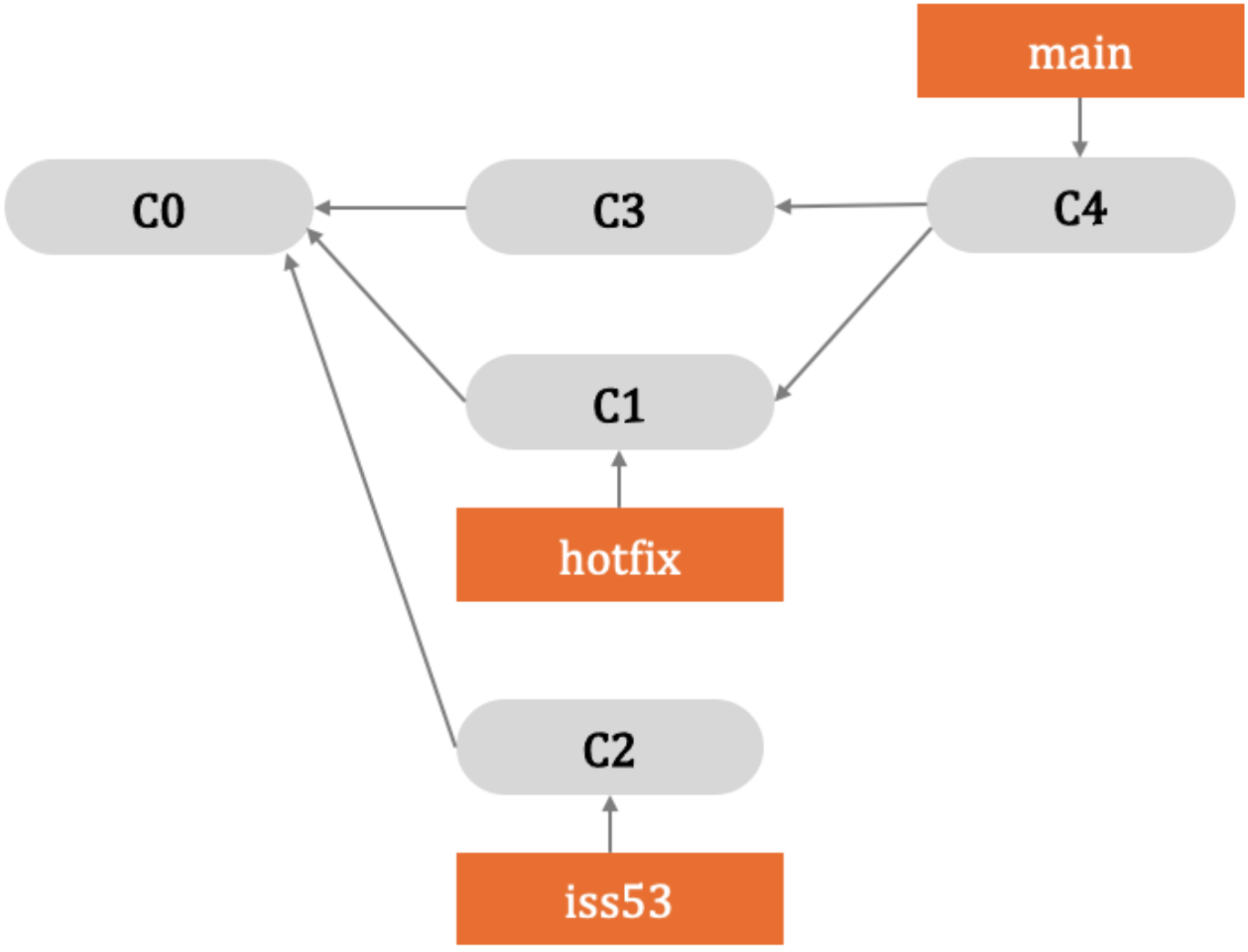
```
git add add.py
git commit -m "C3: resolved conflict between main and hotfi
```

- *Note: if you forget to type a commit message when merging, Git will open up a text editor in Terminal and force you to write a message.*

```
# Please enter a commit message to explain why this merge i
# especially if it merges an updated upstream into a topic
#
# Lines starting with '#' will be ignored, and an empty mes
# the commit.
```

- *To avoid this, always include a commit message*

Three way merge with a conflict VIII



do-pair-share: merge conflicts

Goal: resolve the conflict between `main` and `iss53`

1. *(Already done) On github.com, fork* *`https://github.com/uchicago-harris-dap/merge_example`*
2. *(Already done) Clone your fork to local (can use Github Desktop for now)*
3. Navigate into that folder with Terminal
   - Either use `cd` in command line
   - Or Github Desktop -> Repository -> Open in Terminal
4. Merge `iss53` into `main`
5. Using VSCode, open `add.py` and resolve the conflict in favor of the `iss53` version
6. `git add add.py`
7. `git commit -m "C4: resolved conflict between main and iss53 and merged"`

Resolving file-level conflicts I

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

Resolving file-level conflicts II

Repo

- Branch *iss1* has a version of the file *Example.pdf*
- Branch *main* has another version of *Example.pdf*
- How to resolve the conflict when merging *main* into *iss1*?
- Example **here (link)**

Resolving file-level conflicts III

- Solution: `git checkout` command with
  - `--theirs`: keep file from incoming branch (main)
  - `--ours`: keep file from current branch (iss1)

```
$ git switch iss1
$ git merge main
$ git checkout --theirs Example.pdf
```

- Since I switched into `iss1`, `--theirs` refers to `main`

Resolving file-level conflicts III

- Solution: `git checkout` command with
  - `--theirs`: keep file from incoming branch (main)
  - `--ours`: keep file from current branch (iss1)

```
$ git switch iss1
$ git merge main
$ git checkout --theirs Example.pdf
```

- Since I switched into `iss1`, `--theirs` refers to `main`

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

Summary

- Merging brings changes from one branch into another
  - Easier case: *fast forward* when there is no diverging
  - Harder case: *three way merge* when there is diverging
- 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

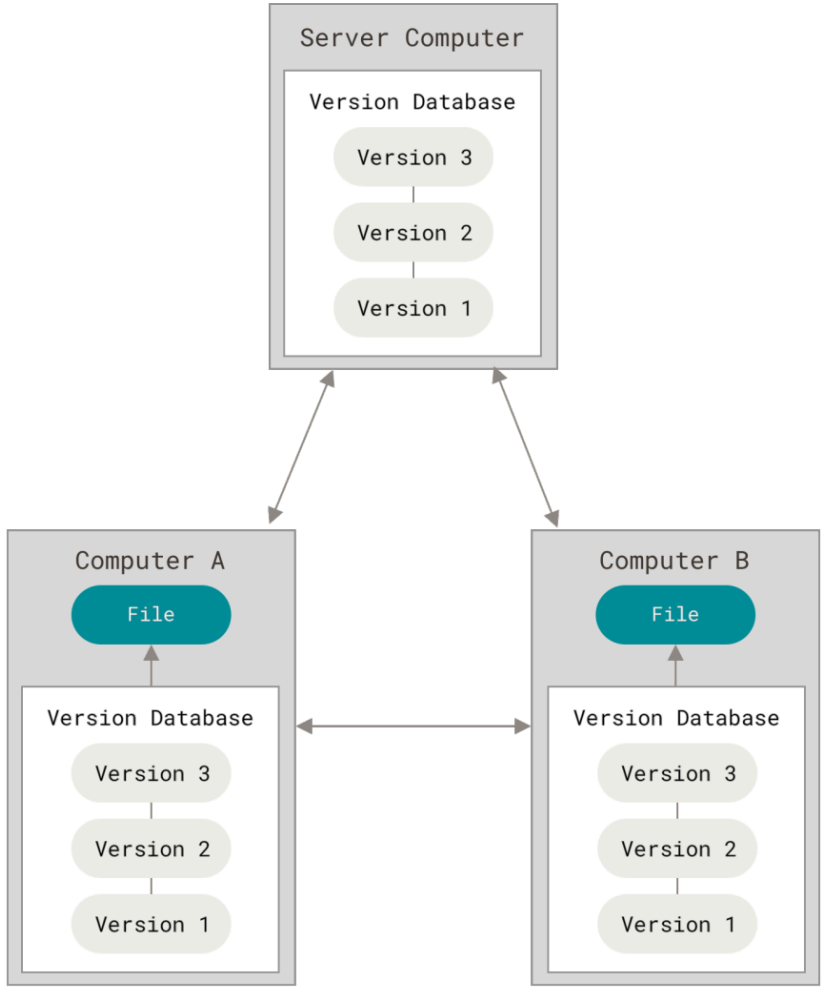
- push
- fetch/pull
- list all branches
- fetch
- pull
- pull requests
- push

Remote: Github

- Up to this point, everything we've been doing is local
- But for collaboration, you will be working off a shared directory
- And that shared directory is usually hosted on Github

“Distributed” version control

- Each computer fully mirrors the remote repository, including its full history



push

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

- push (committed) changes from local to remote repository

fetch

```
$ git fetch
```

- Fetches changes from remote repository to your computer – but doesn't merge then in yet
- This command does not modify any of your existing work.
- If you did some work in the same branch, you will need to resolve conflicts with the remote
  - *git merge origin*

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.
- Remark: “sync changes” (appears in GH desktop and VSCode) really means push and pull

see *all* branches

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

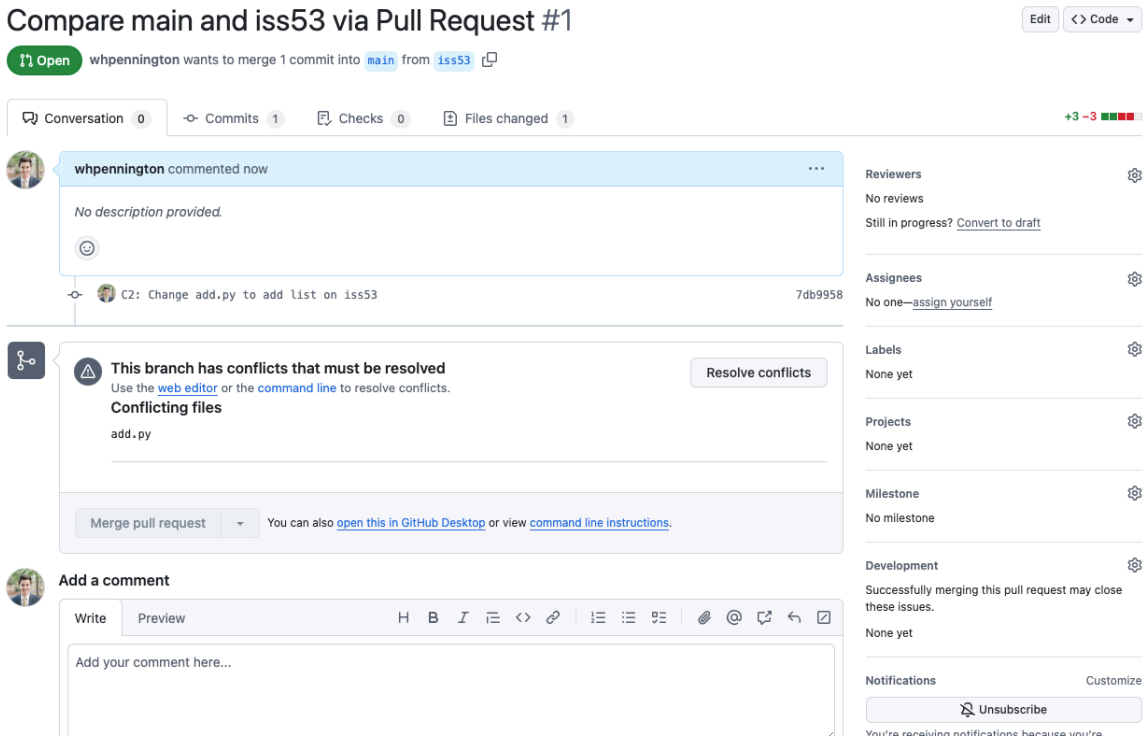
```
$ git branch -a
```

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."

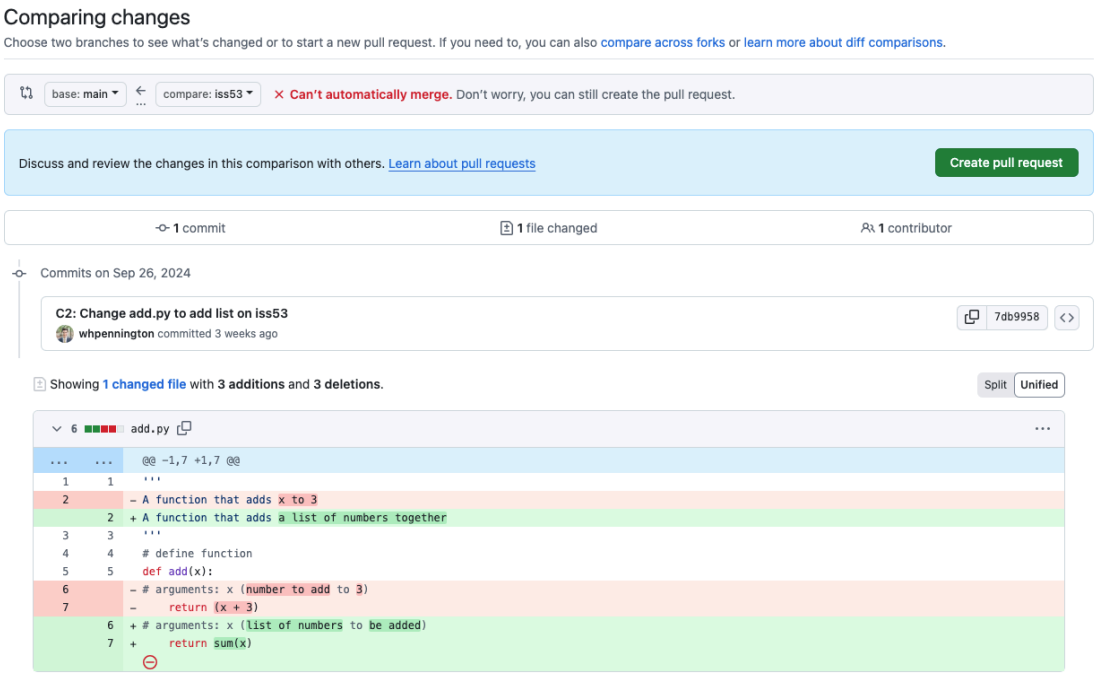
Pull Requests

- A pull request is a way to propose changes from your branch to be merged into the main project/branch
- Doesn't `git merge` already do this? Yes, but pull requests allow teammates to review your code before it gets merged.
- 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`

Summary

- *push* sends changes from local to remote
- *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

Final Remarks

- Git is admittedly confusing!
- Now that you know how it works, you have to start using it to get comfortable using the commands
- MS: I have a printed version of a Git cheatsheet (`atlassian_cheatsheet.pdf`) taped on the wall in my office