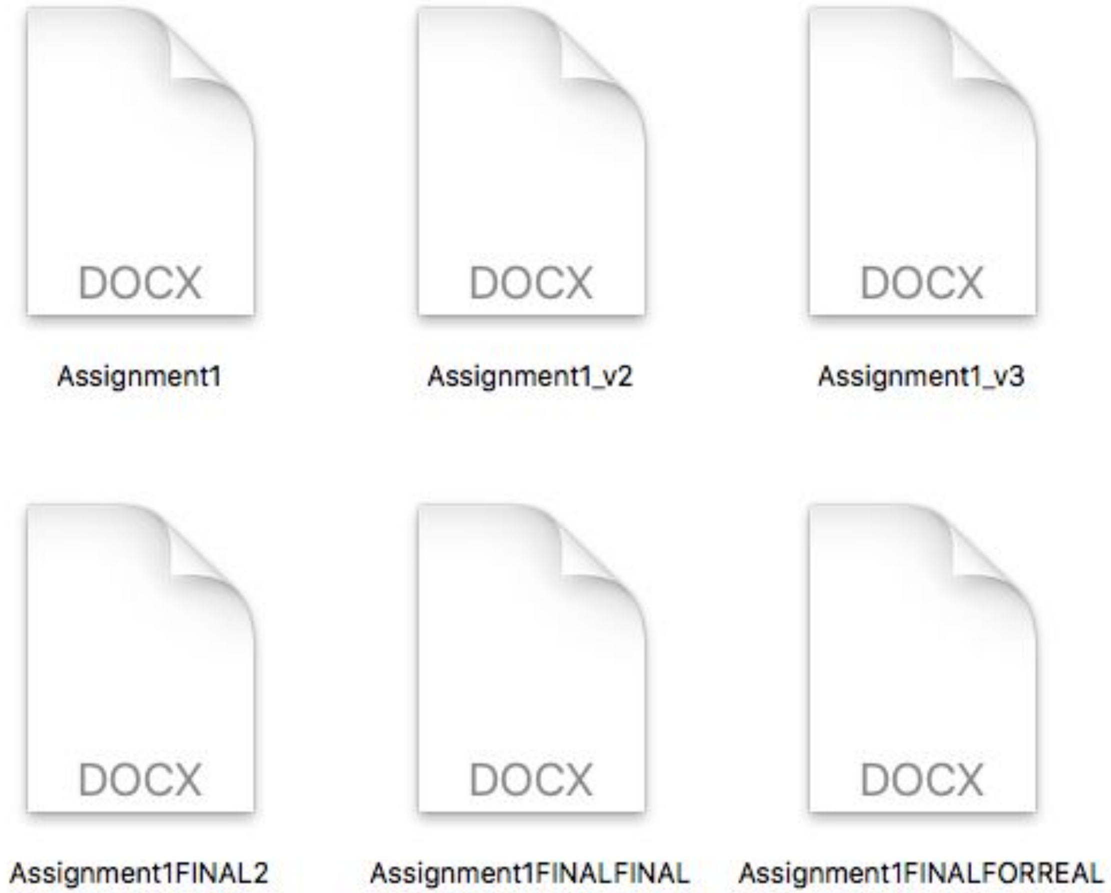
presents...

GIT EDUCATED with diversIT!

An introductory workshop on version control.

WHAT IS VERSION CONTROL?


DOCX
Assignment1
DOCX
Assignment1_v2
DOCX
Assignment1_v3
DOCX
Assignment1FINAL
DOCX
Assignment1FINAL2
DOCX
Assignment1FINALFINAL
DOCX
Assignment1FINALFORREAL

Version control is a system for keeping track of:

1. **<u>When</u>** the file was modified
2. **<u>What</u>** was modified
3. **<u>Why</u>** it was modified
4. **<u>Who</u>** modified the file

Version control enables users to **<u>restore previous versions</u>**.

WHAT IS GIT?

Git is a piece of version control software (VCS).

- Most **widely used** modern VCS
- Performs all previously mentioned functions of **version control**
- Open source

Other VCS include **Mercurial**, **Apache Subversion (SVN)** and **Team Foundation Server (TFS)**.

Version control using git
@NPCompleteTeens
Version control using codeshare.io
Version control using google docs
Version control by having one person maintain the master locally and sending them code on messenger so they can merge manually

[Source: Nondeterministic Memes for NP Complete Teens]

WHAT IS GITHUB?

GitHub is a Git repository hosting service.

- Hosts **remote** repositories using Git
- Enables easy collaboration in software teams
- Includes more features like **forking**, **pull requests** and **issue tracking**

Other Git repository hosting services include **Bitbucket** and **GitLab**.

What we're covering in the demo:

- ❖ Setting up a repository
- ❖ Saving changes
- ❖ Stashing changes
- ❖ Branches and merging
- ❖ Undoing changes
- ❖ Collaborating (with GitHub)

SETTING UP A REPOSITORY

1. Create a local repository

```
$ git init
```

2. Display or change the configuration

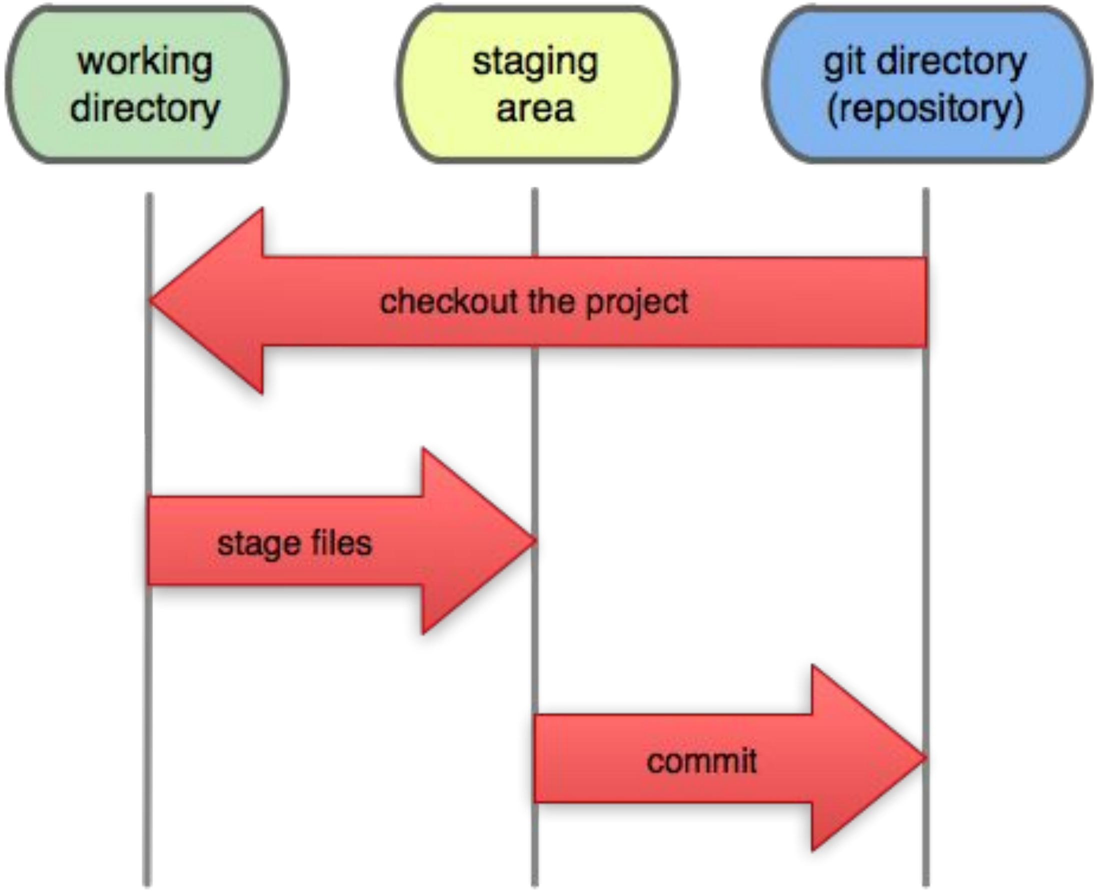
```
$ git config --system user.email
```

```
$ git config --global user.name
```

```
$ git config --local user.name “<name>”
```

SAVING CHANGES

Local Operations



[Source: Stack Overflow]

3. Add a new file in the working directory

4. Display the state of the working directory and staging area

```
$ git status
```

5. Stage the new file to be committed

```
$ git add <file>
```

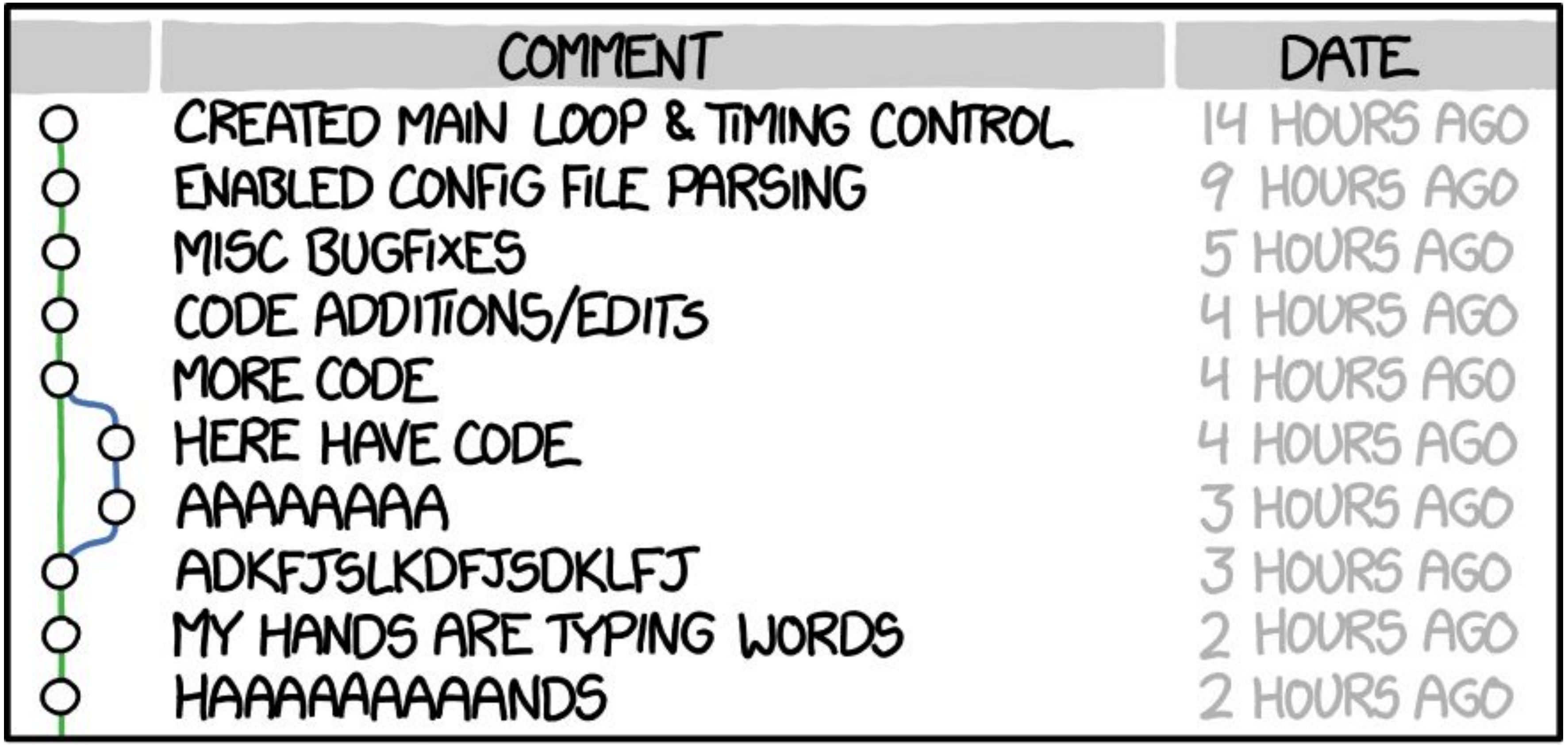
```
$ git add <directory>
```

```
$ git add .
```

Stage all changed files

6. Commit the staged changes

```
$ git commit -m “<message>”
```



	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

[Source: XKCD]

7. Add, stage and commit a .gitignore file

```
$ git add .gitignore
```

```
$ git commit -m “Add .gitignore”
```

Git categorises each file in the working directory as one of three states:

1. **Tracked** - has been previously staged or committed
2. **Untracked** - has not been staged or committed
3. **Ignored** - a file which Git has been told to ignore

We can tell Git which files to ignore with a .gitignore file.

What do we usually want to ignore?

- Dependency caches (/packages)
- Compiled code (.o, .pyc and .class)
- Build output directories (/bin)
- Files generated at runtime (.log, and .tmp)
- Hidden system files (.DS_Store)
- IDE config files (.idea/)

8. Change the name of a file and stage

```
$ git mv <old file name> <new file name>
```

9. Delete a file in the working directory and stage

```
$ git rm <file>
```

10. Remove a file from git but keep the file in the directory

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

STASHING CHANGES

11. Stash tracked changes away and revert the working directory

```
$ git stash
```

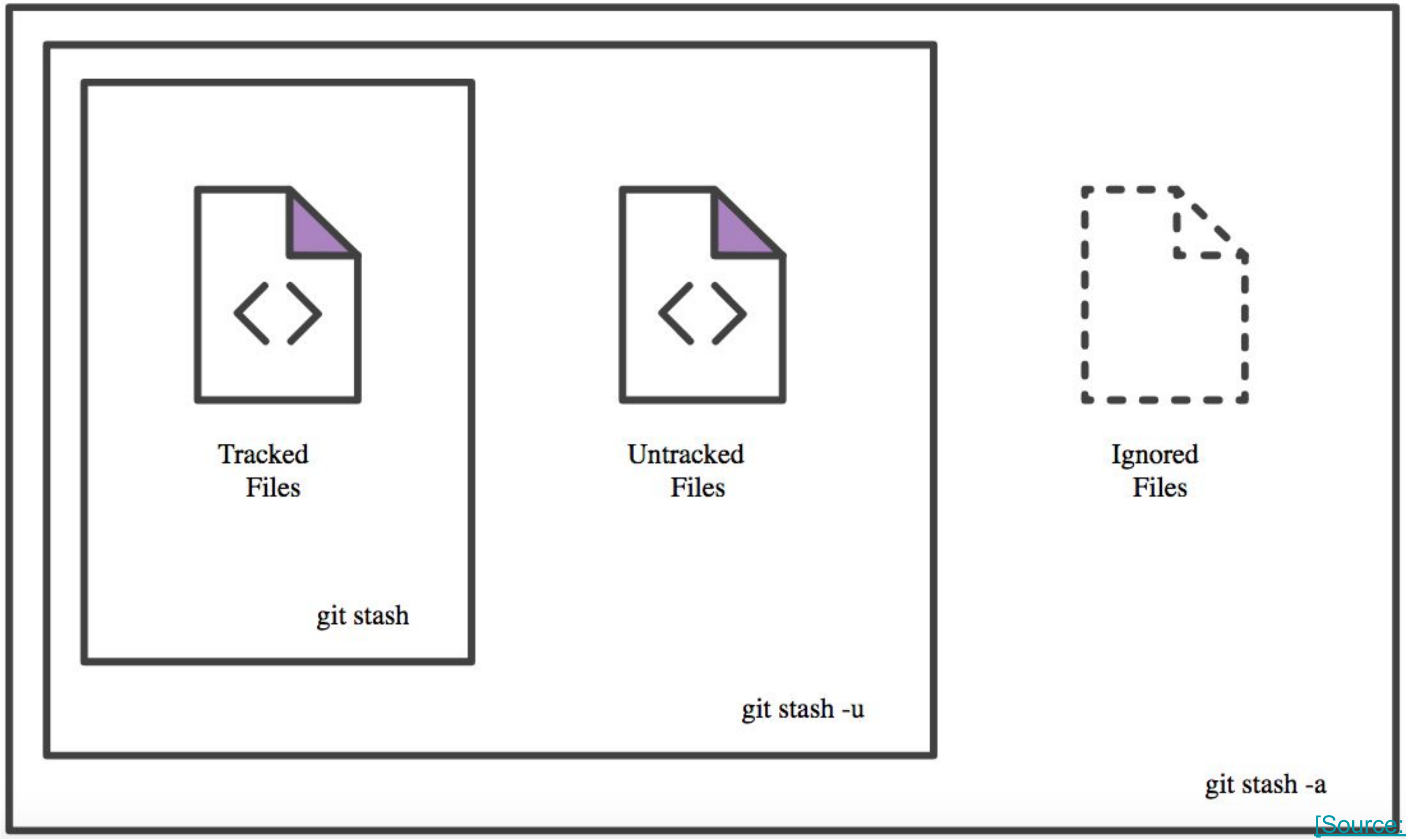
12. Re-apply the stashed changes and remove from stash

```
$ git stash pop
```

13. Re-apply the stashed changes and keep in stash

```
$ git stash apply
```

git stash options



[Source: Atlassian]

BRANCHES AND MERGING

14. Create a branch

```
$ git branch <branch>
```

One feature = one branch

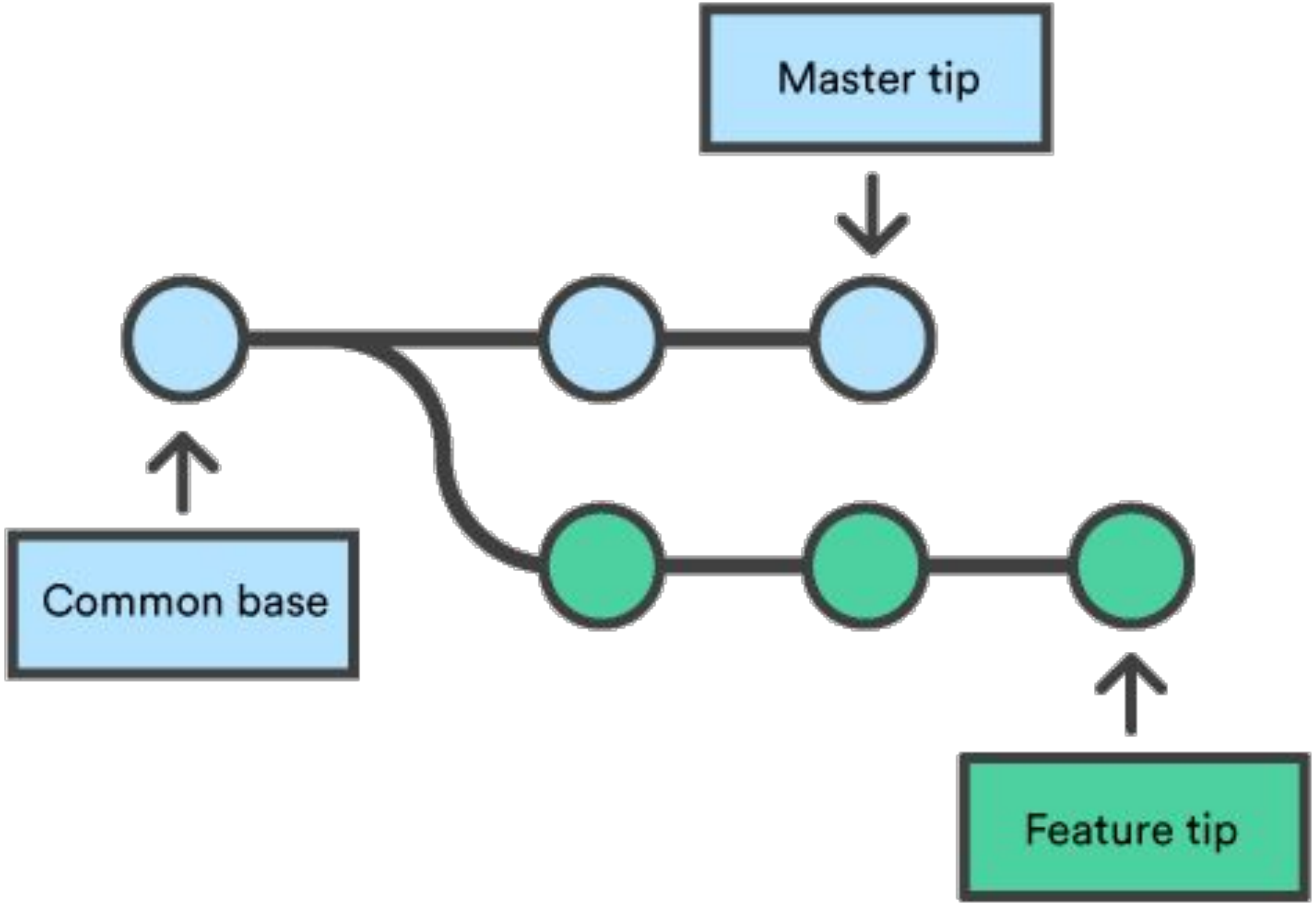
15. Make and commit some change in the master branch

16. Switch to the new branch

```
$ git checkout <branch>
```

17. Modify the file in the working directory

18. Commit the staged changes in the new branch



[Source: Atlassian]

19. Switch back to the master branch

```
$ git checkout master
```

20. Merge the new branch with the master branch

```
$ git merge <branch>
```

21. Resolve the merge conflict

```
$ git mergetool
```

When you see your project partner about to push code that will cause a inevitable merge conflict when you eventually push

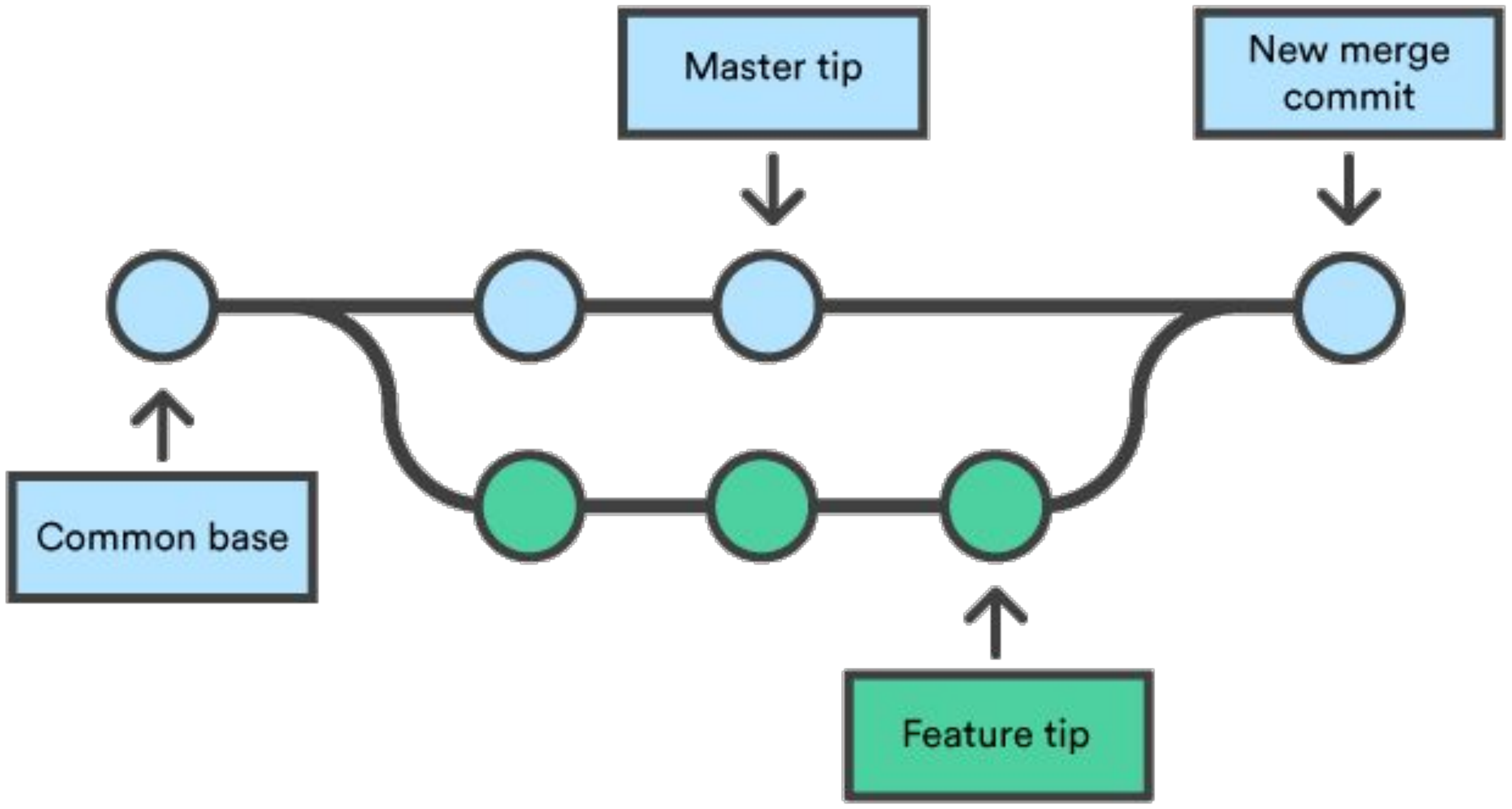
[Source: Nondeterministic Memes for NP Complete Teens]

22. Commit the merge

```
$ git commit -m “Merge <branch> with master”
```

23. Delete the new branch

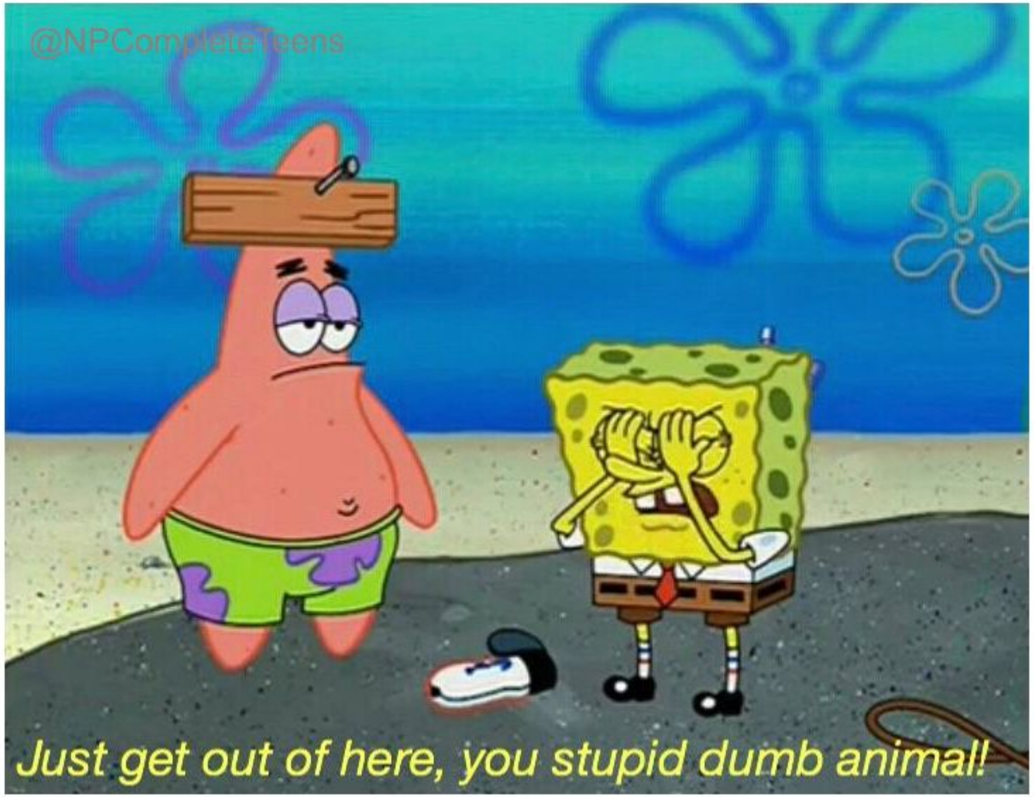
```
$ git branch -d <branch>
```



[Source: Atlassian]

UNDOING CHANGES

When your code is so f[redacted]ed up you have to hit it with the “`git reset –hard HEAD`"



[Source: Nondeterministic Memes for NP Complete Teens]

24. Display the commit log

```
$ git log --oneline
```

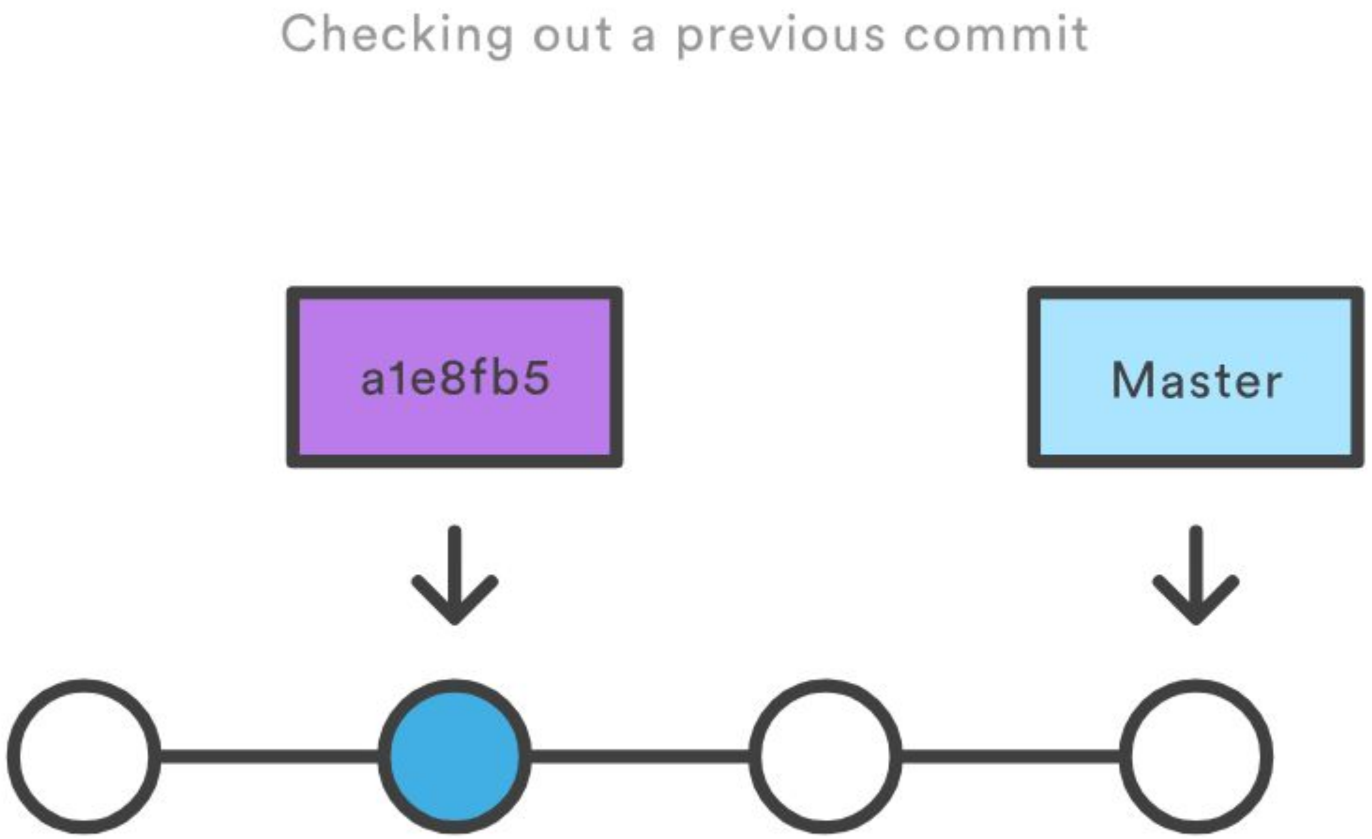
25. Review an old commit

```
$ git checkout <commit>
```

Detached HEAD state

26. Create and switch to a new branch where future commits don’t exist

```
$ git checkout -b <branch>
```



[Source: Atlassian]

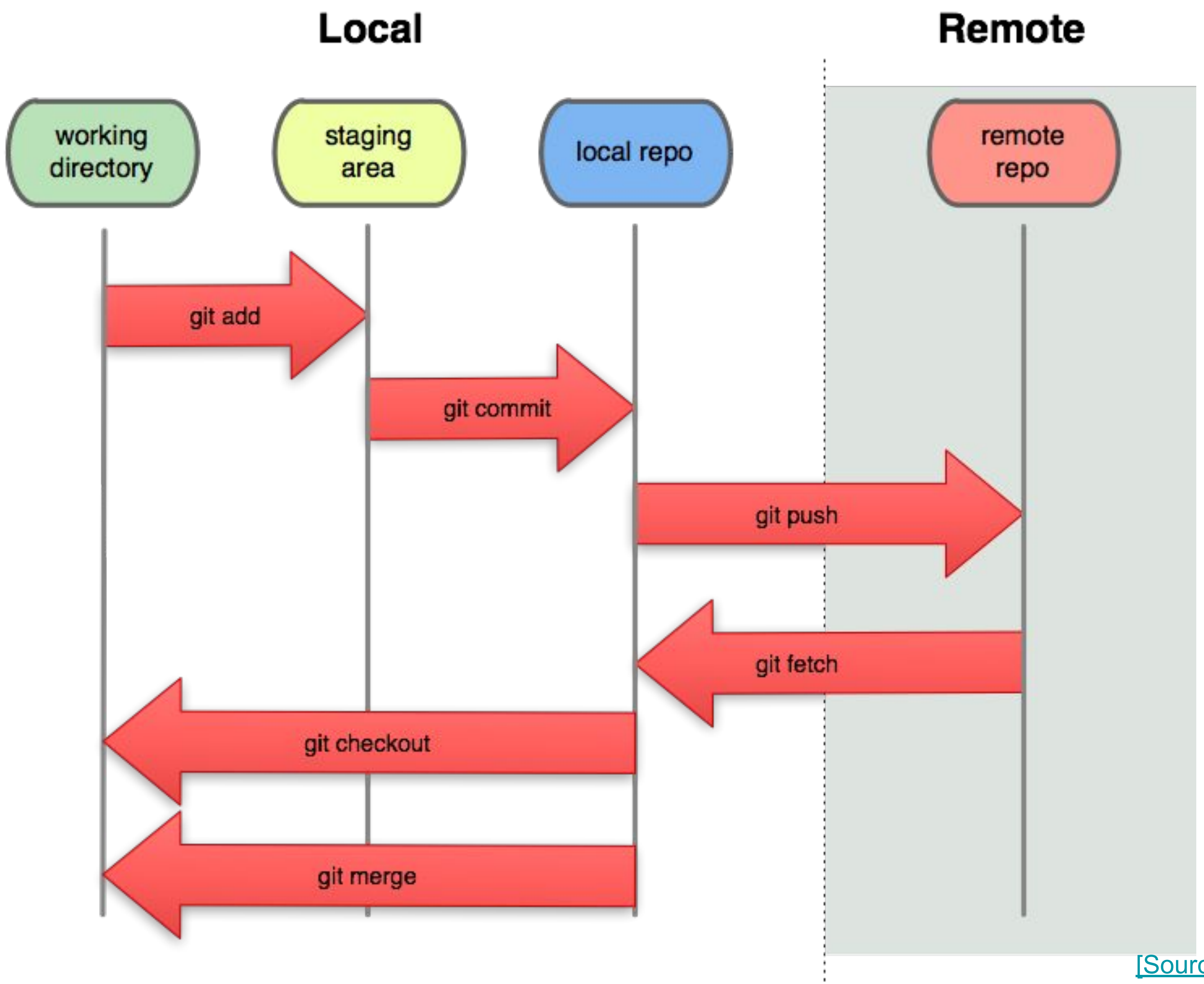
27. Create a new commit that undoes the previous one

```
$ git revert <commit>
```

28. Change the commit history

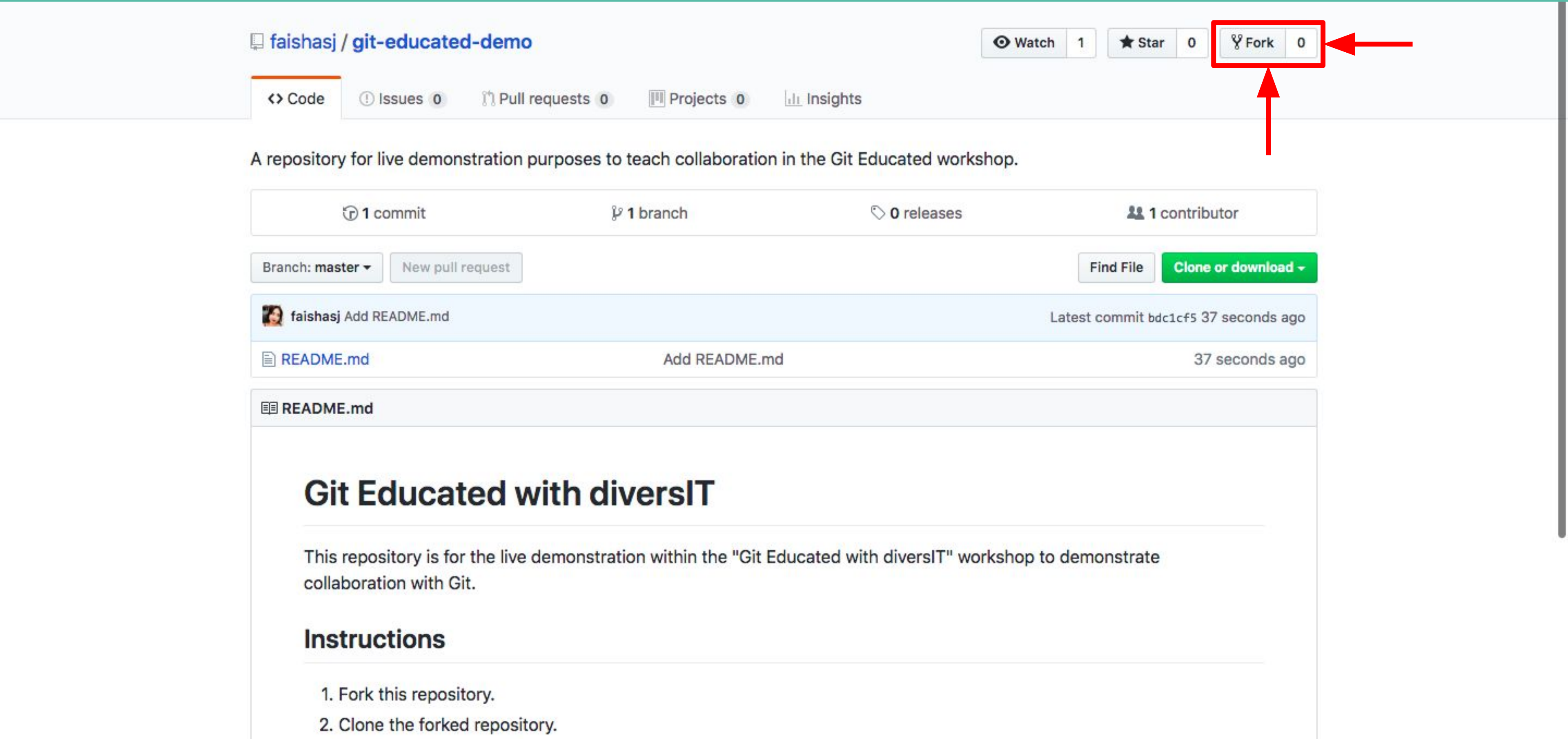
```
$ git reset <commit>
```

COLLABORATING



[Source: Statistical [R]ecipes]

29. Fork the repository on GitHub



A repository for live demonstration purposes to teach collaboration in the Git Educated workshop.

faishasj Add README.md Latest commit bdc1cf5 37 seconds ago

README.md	Add README.md	37 seconds ago

README.md

Git Educated with diversIT

This repository is for the live demonstration within the "Git Educated with diversIT" workshop to demonstrate collaboration with Git.

Instructions

1. Fork this repository.
2. Clone the forked repository.

30. Clone the forked repository

```
$ git clone https://github.com/<username>/git-educated-demo.git
```

31. Add an upstream remote

```
$ git remote add upstream https://github.com/faishasj/git-educated-demo.git
```

32. Fetch and pull changes from the upstream remote

```
$ git fetch upstream <branch>
```

```
$ git pull upstream <branch>
```

33. Create and checkout a new branch

```
$ git checkout -b <branch>
```

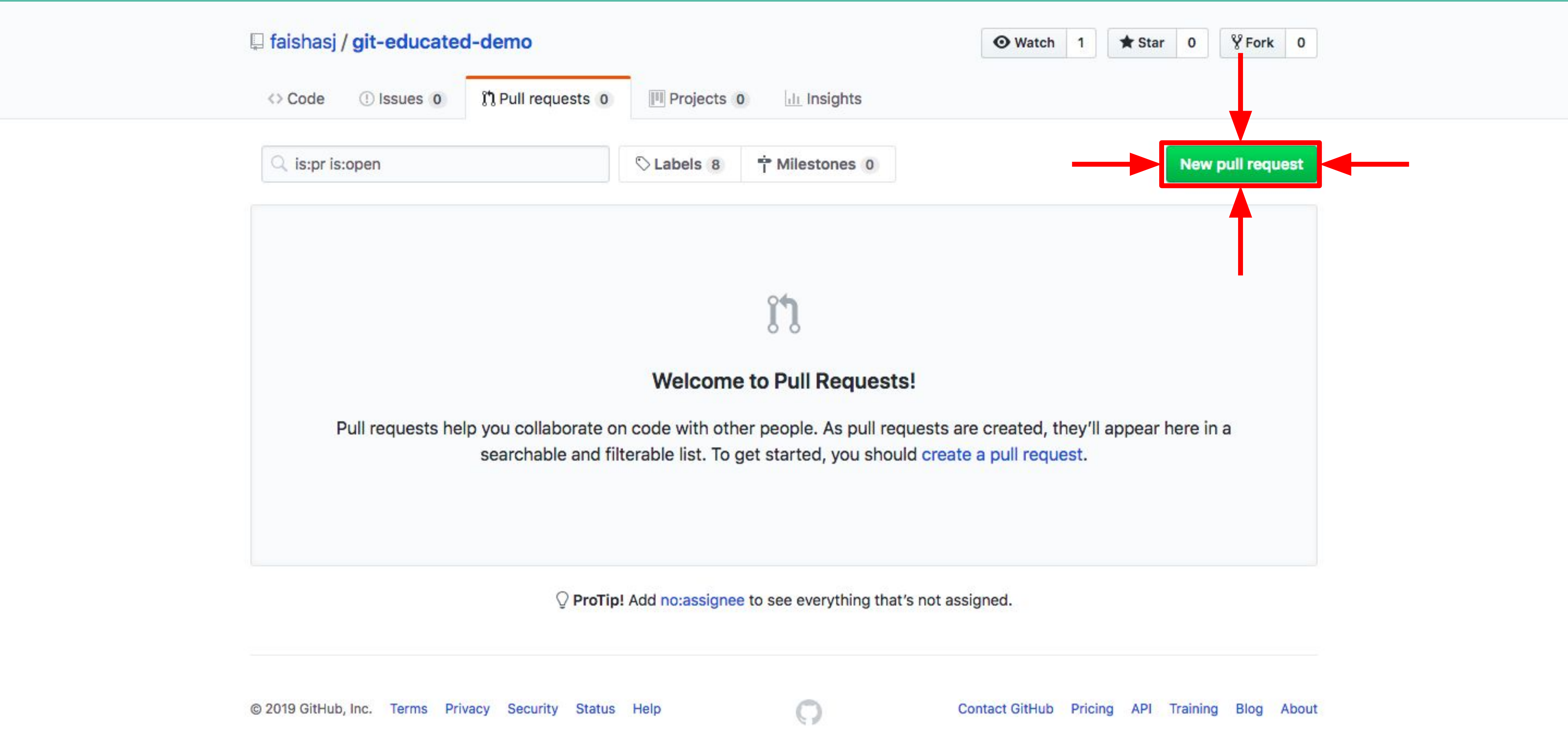
34. Add your name under the correct date in attendees.

35. Commit and push changes to the origin remote.

```
$ git commit -m “<message>”
```

```
$ git push origin <branch>
```

36. Submit a pull request



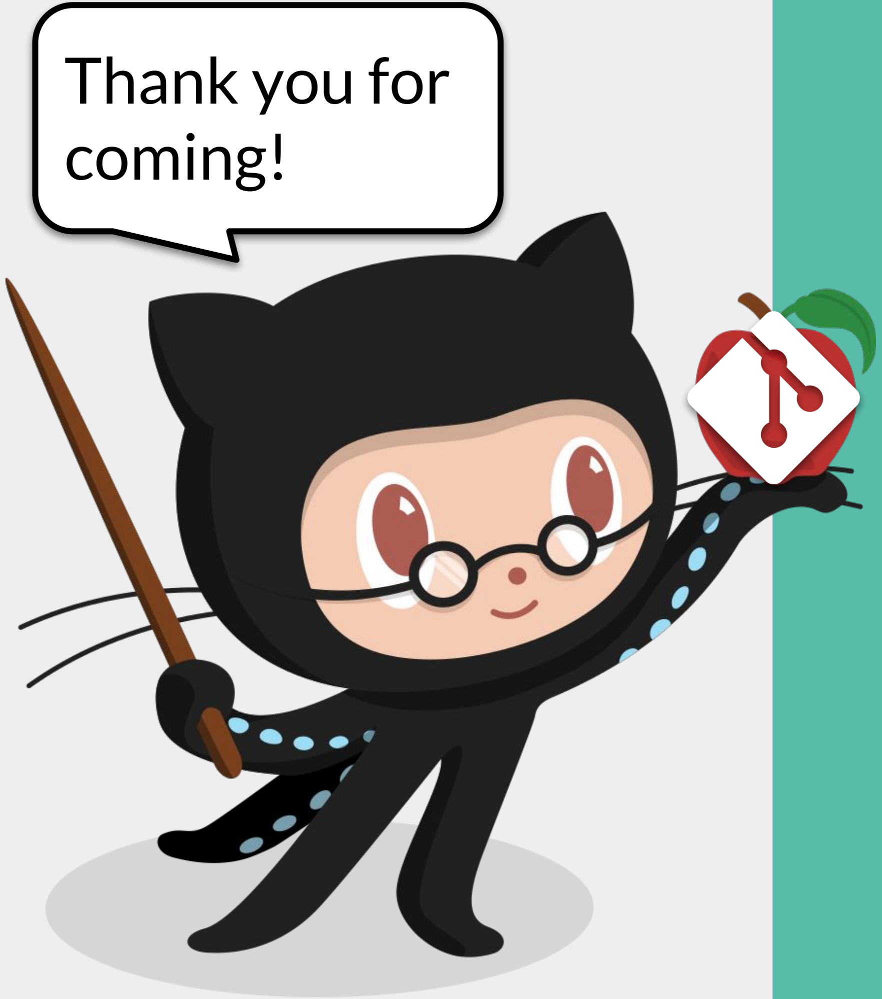
Welcome to Pull Requests!

Pull requests help you collaborate on code with other people. As pull requests are created, they'll appear here in a searchable and filterable list. To get started, you should create a pull request.

ProTip! Add no:assignee to see everything that's not assigned.

How engineers be after committing one line of code to an open source project

[Source: Nondeterministic Memes for NP Complete Teens]



What we've covered today:

- Setting up a repository
- Saving changes
- Stashing changes
- Branches and merging
- Undoing changes
- Collaborating (with GitHub)