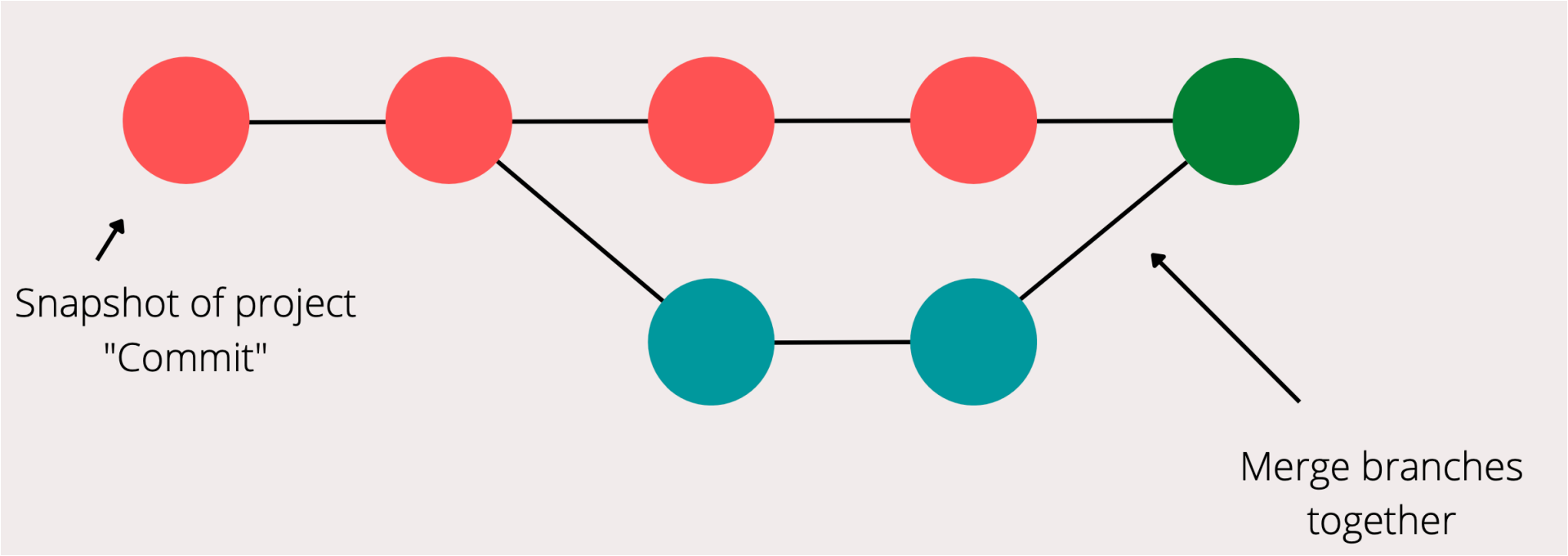
Intro to Git & Github

Version Control

- Version control is a system that records changes to a set of files over time, allowing you to recall specific versions later
- More elegant and formal replacement for a common solo informal version control (i.e., “file_v1.txt”, “file_v2.txt”, “file_final.txt”, “file_really_final.txt”)



WHAT IS GIT?

Git is a Distributed Version Control System

It manages and tracks changes over time for all the files in a particular project

In Git vocabulary, each individual project is called a "Repository"

Git was developed in 2005 as the version control system for Linux

Today it is widely used and the most popular version control systems in the world

WHAT IS GITHUB?

GitHub is a web based hosting service for Git repositories

It has similar distributed version control functionality to Git, while also adding it's own features

GitHub was developed in 2007 and was sold to Microsoft in 2018

GitHub is commonly used to host open source software development and data science projects

≠

- Runs locally on your machine
- Internet not needed
- No account needed

- Web based service that hosts Git repositories in the cloud
- Makes it easy to share and collaborate with others
- Account and internet required

Git and the command line

Git was built for the command line

- Git is primarily a command line tool
  - although there are many tools available that allow you to avoid the command line
- Using git in the command line will always give you the most flexibility
- We will use the basics from the command line
- Then we will see how to use git in VS Code

ESNT's Command Line Cheat Sheet

`pwd`	print path of current working directory
`cd path/of/target`	change directory to "target" directory
`cd ~`	change directory to the home directory
`cd ..`	change directory one level up
`ls`	directory listing
`ls -la`	long directory listing and display of hidden files
`ls | wc -l`	count the number of files in a directory
`cat file.txt`	displays the contents of a file
`less file.txt`	displays the contents of a file one page at a time (press `q` to exit 'less' mode)
`mkdir my_directory`	create a new directory
`touch file.txt`	create an empty file called 'file.txt' (or updates timestamp if file already exists)
`nano file.txt`	Edit a file with the nano editor
`rm file.txt`	delete a file (careful this is permanent)
`rm -rf my_directory`	delete a directory recursively (careful this is permanent)
`mv old_name.txt new_name.txt`	change the name of a file
`mv file.txt new/file/path/`	move a file to new/file/path directory
`du -h`	display size of directory and all subdirectories in "human" units
`du -h -d 1`	display size of directory and one level of subdirectories
`du -h -d 1 | sort -h`	same as previous line but sorted by size
`history`	(bash) display the history of the terminal
`history 0`	(zsh) display the whole history in zsh shell
`clear`	clear the terminal (doesn't delete any history)

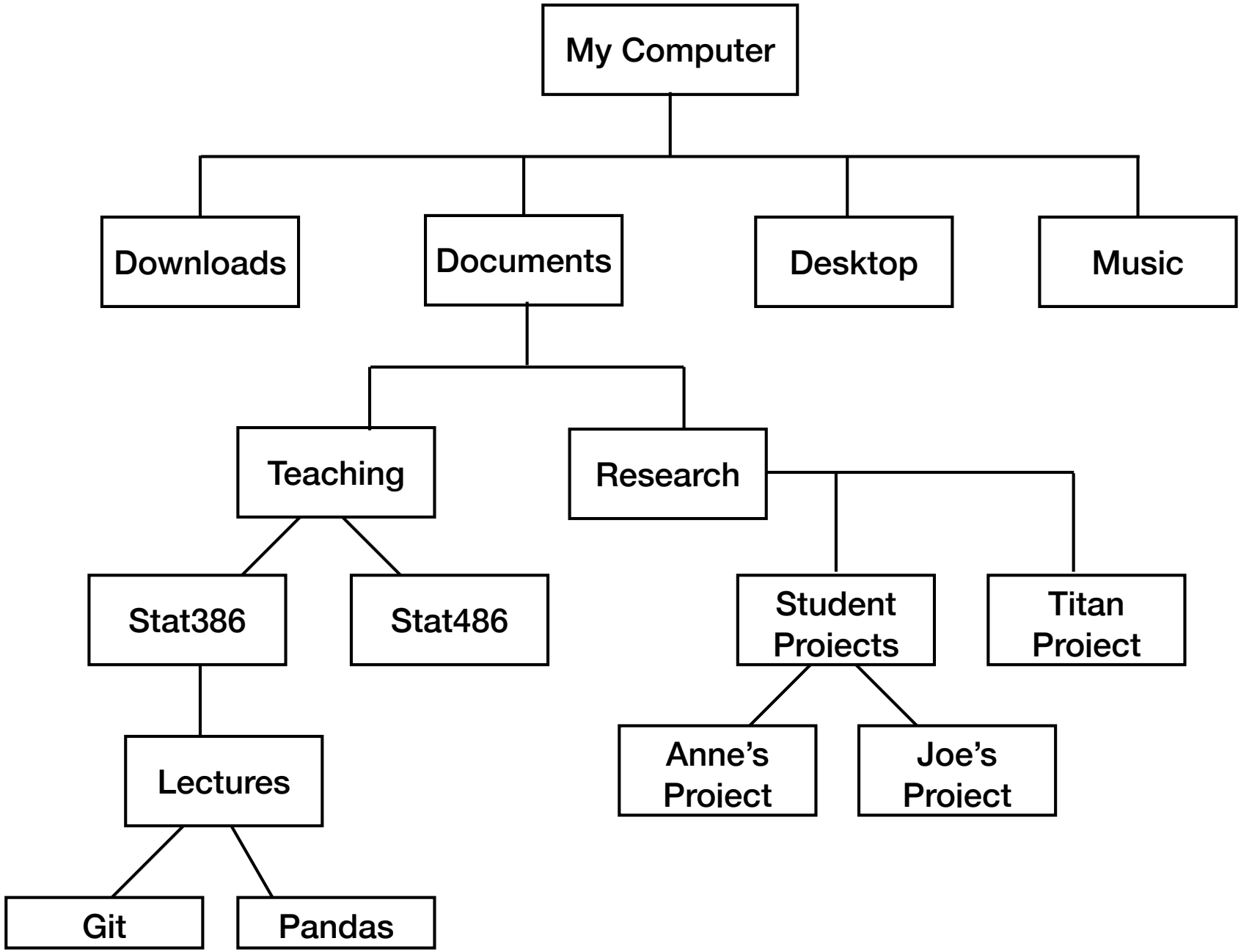
Most common commands

- By far, the most common commands that I use:
  - `cd`
  - `ls` (or dir in Windows)
  - `git` commands

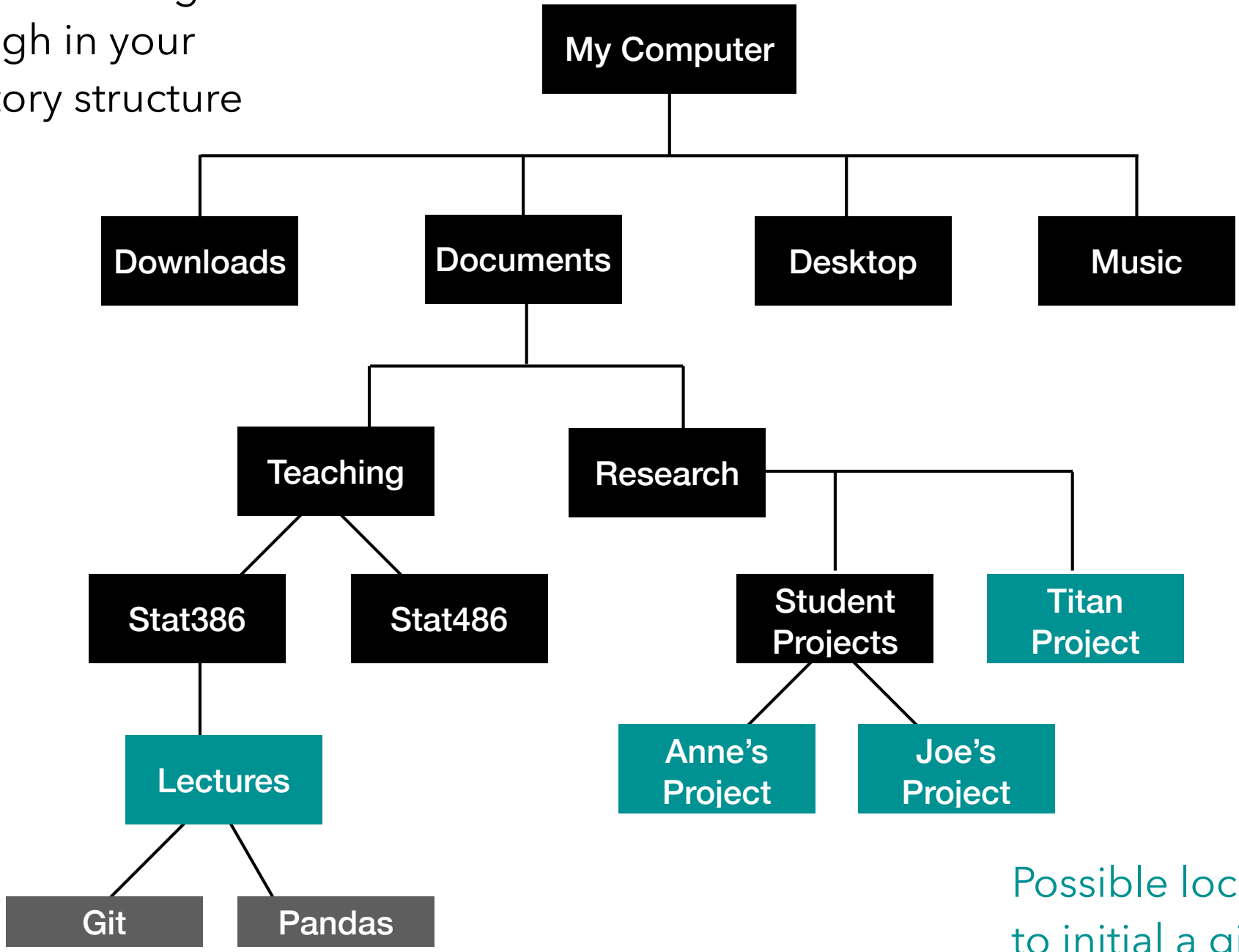
Git overview

Understanding git

- The main “unit” that git tracks is a **repository**
- A repository is basically a folder containing all the files relevant for a **single** project
- Even though git is installed on all your computers, we must tell git when we want it to keep track of a particular repo
  - Each separate project / repository must be initialize separately
  - NEVER initialize git for the entire set of folder and files on your computer


My Computer
Downloads
Documents
Desktop
Music
Teaching
Research
Stat386
Stat486
Student Proiects
Titan Proiect
Lectures
Anne's Proiect
Joe's Proiect
Git
Pandas

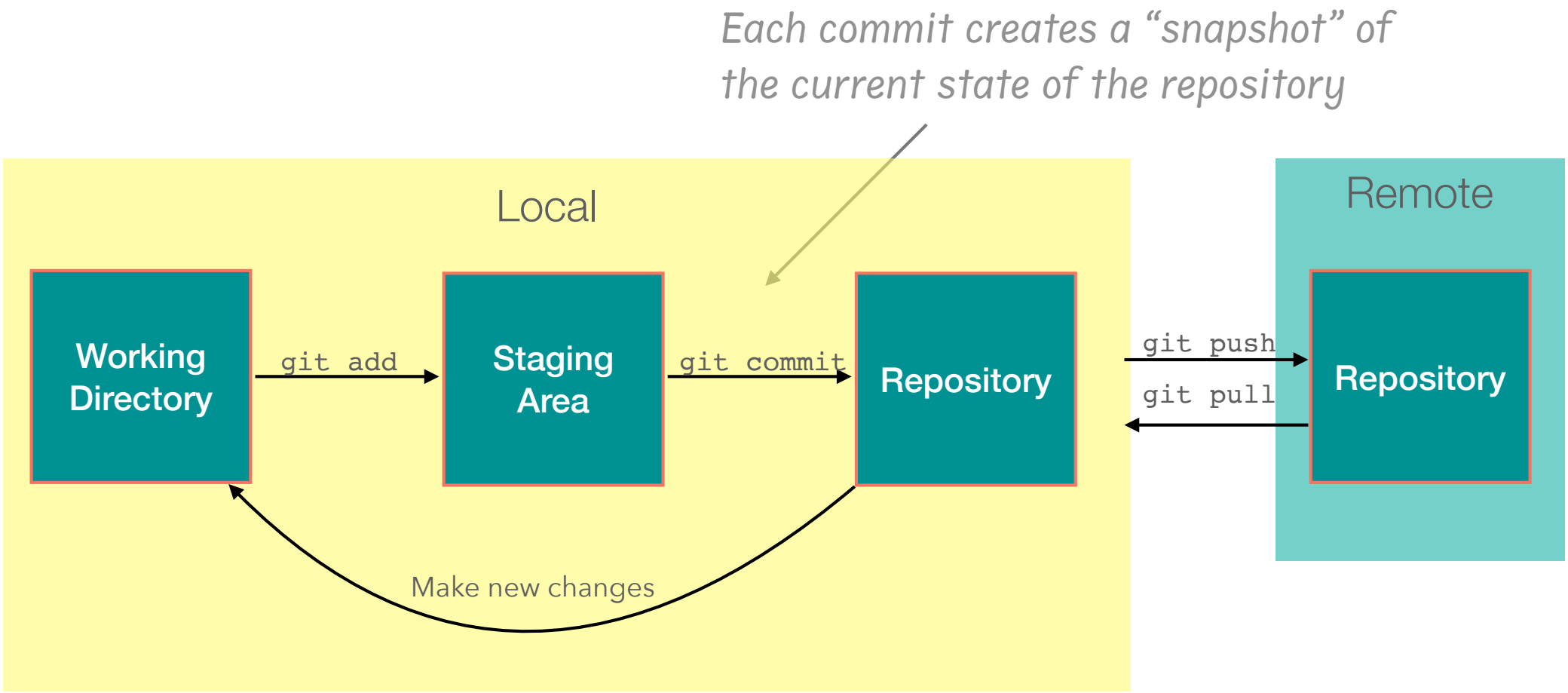
Don’t initialize git too high in your directory structure



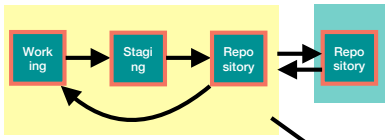
Don’t initialize git repositories inside other git repositories

Possible locations to initial a git repository

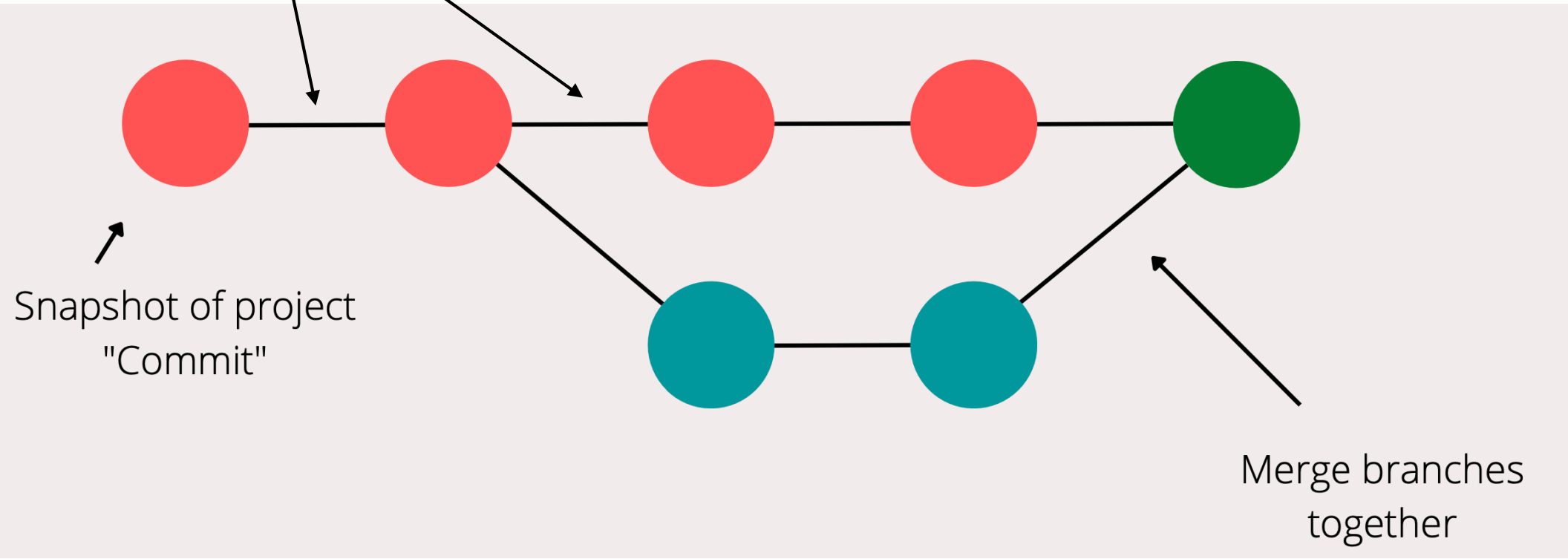
Basic Git Workflow



Basic Git Workflow



In between each commit, we make changes to the repo & then stage the changes



Getting started with git

Configure git on your local machine

- Before you start using git on your local machine you must configure some of the settings
- This only needs to be done **once** (do it today and don't have to do it again)
  - Required:
    - name
    - email
  - Optional (but required for this class):
    - default editor
    - default branch name

Configure git on your local machine

- Name:
  `git config --global user.name “First Last”`
- Email:
  (Use email associated with GitHub account)
  `git config --global user.email “email@example.com”`
- Default editor:
  (Windows users might have already done this during installation)
  (This example sets VS code to be editor – other choices are possible)
  `git config --global core.editor “code --wait"`
- Default branch name:
  `git config --global init.defaultBranch main`

Mac users might also need this step

- In Visual Studio Code
  - cmd + shift + p
  - Search “code”
  - Click on “Shell command: install command ‘code’ in PATH

Just in case: default editor is vi

- If you forget to add a commit message (and type `git commit` without the `-m`), the default editor will pop up for you to add a commit message.
- The default editor is "vi"
  - hit the "i" key (to go into interactive mode)
  - type the commit message
  - to exit: hit "esc" then type `:wq`

Starting a git repository

- In git vocabulary, a "repository" is a folder containing all the files and subfolders pertaining to a single project
- When we start or initial a repo, we are telling git to start tracking changes in the folder
- To start git tracking in a blank or pre-existing folder:
  - `git init`
  - *Note:* `git clone <url>`*: is a way to copy a remote repo onto the local machine. Cloned repositories will automatically be setup to use git*

Reminder

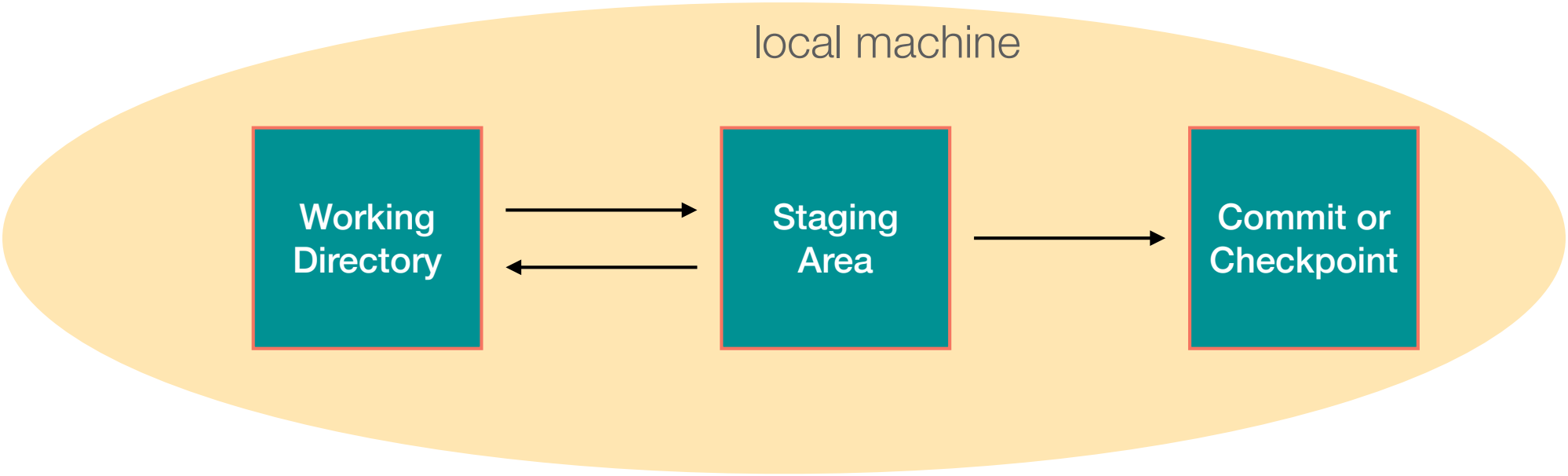
- The command git init should be run in the top directory of the project
- CAUTION:
  - Do not initial git repositories inside git repositories
    - (Don't run "git init" in a subfolder of an existing git repository)
  - Do not run "git init" in your root directory
  - For each project, "git init" only needs to be run once to set up git tracking

What happens after running “git init”

- Running “git init” creates a git repository
  - Meaning that we are telling git to start tracking changes inside this directory / repository
- Behind the scenes git creates a hidden “.git” folder with all the tracking data
- Git tracking can be removed by deleting the .git folder
  - The git repo will go back to being a regular untracked folder
  - `rm -rf .git`

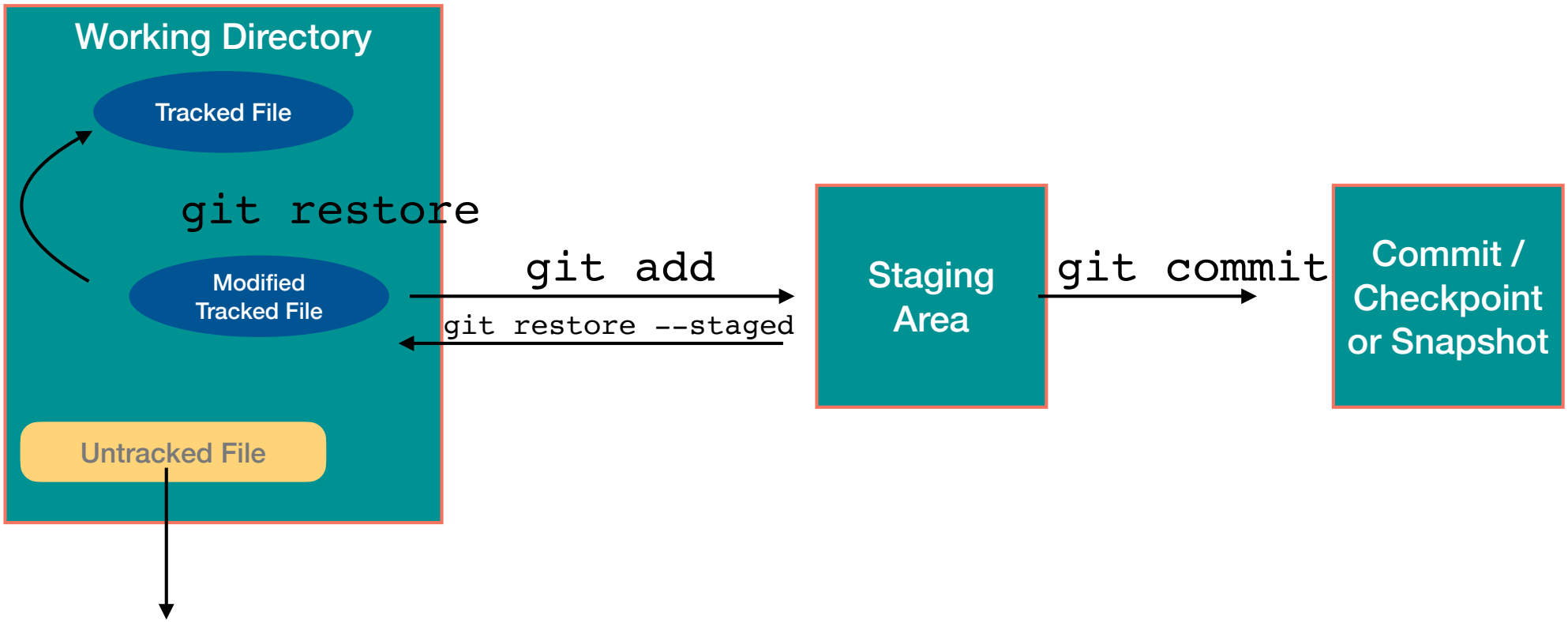
Git Environments

Git file environments or states



FILES STATES

- Tracked
  - Unmodified (version in WD same as latest commit)
  - Modified (version in WD changed from latest commit)
  - Staged (modified version in staging area)
- Untracked



Git doesn't monitor untracked files

New files must be staged then committed to become tracked

Common commands

git Workflow. *Commands for the typical workflow.*

`git status`
show modified files and files in staging area

`git add [file]`
add [file] to the staging area

`git add .`
add all changed files to the staging area

`git commit -m "[commit message]"`
commit all files in the staging area

`git commit -a -m "[commit message]"`
add all changed files and commit in one step

`git log`
show all commits in the current branch's history

`git log --oneline`
show a shorter version of all commits in branch history

Commit Messages

- Standardize your commit message structure
- Don’t end commit message summaries with punctuation
- Use imperative verb form
  - “Update” instead of “Updated”
  - Think: “If applied, my commit will: _____”
- Be specific – clearly describe what the commit message does
- Include relevant information (for example if you are responding to a pull request or issue)
- If necessary, add a description along with a summary

Try to avoid commit messages like this

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

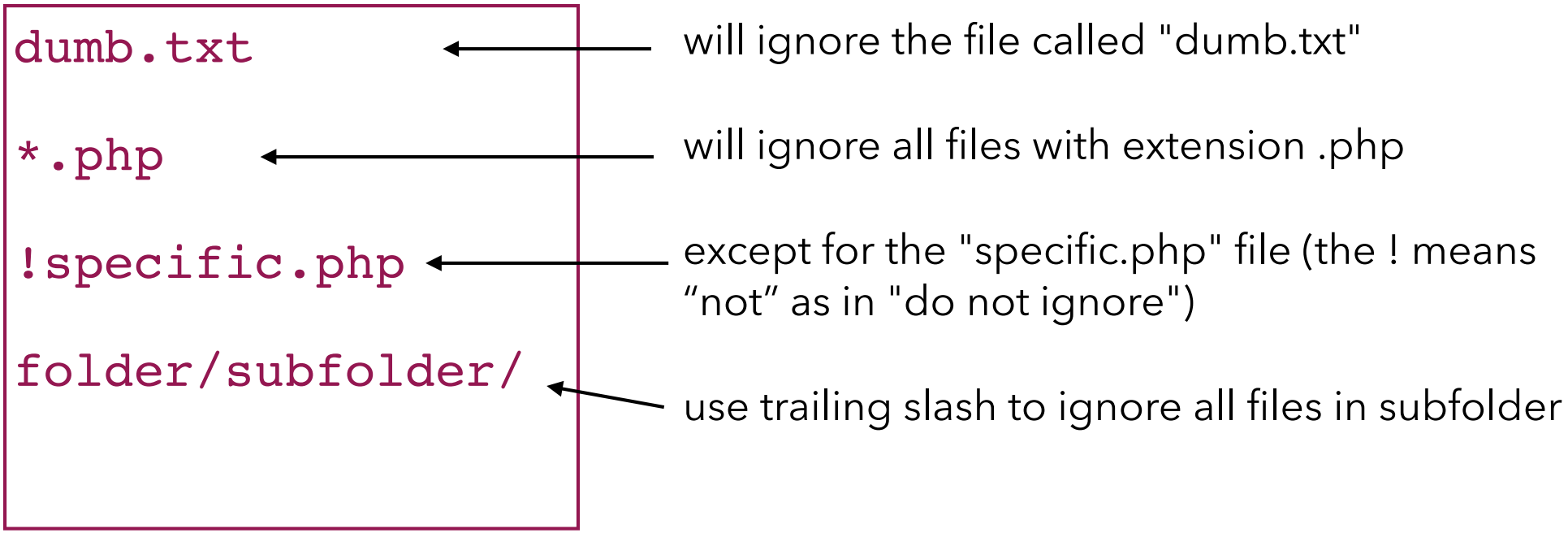
https://xkcd.com/1296/

The .gitignore file

Ignoring files

- Sometimes there are file in the repository that we don't want tracked
- We can ignore files by creating a .gitignore file
- The .gitignore should be in the main folder of the repository
- It can contain names of specific files, specific folders, or patterns
- .gitignore is a specific filename that git recognizes

Example .gitignore file



Common .gitignore for me

```
.DS_Store

.ipynb_checkpoints/

Api-key.txt

Data/
```

IMPORTANT!!

- Add all files that you want to ignore to the .gitignore file BEFORE any of the ignored files are committed
- Git will still track and push files that were committed before they were added to the .gitignore
- It is a good idea to create a .gitignore file as early as possible with files & directories you want to ignore

  For example, make it a habit in this class to add
  `.ipynb_checkpoints/`
  to your .gitignore at the creation of any repository
- It is possible to delete committed files that you want to ignore, but it is easier to just ignore them off the bat (https://www.atlassian.com/git/tutorials/saving-changes/gitignore)

Helpful links

- https://help.github.com/articles/ignoring-files
- https://github.com/github/gitignore

Branches and Merging

Branching

- Branches allow us to create new versions of the project without changing the "main" project
  - Experiment with adding features
  - Team work (your part is done on a branch)

Git branching commands

Branch & Merge. *Creating and merging branches.*

`git branch`
list all branches. (∗) indicates current active branch

`git branch [branch-name]`
create a new branch at current HEAD *(does not switch to new branch)*.

`git switch [branch-name]`
switch active branch to [branch-name]

`git switch -c [branch-name]`
create a new branch and switch to it in one step

`git merge [branch-name]`
merge [branch-name] into the active branch

Aside: Git checkout

- `checkout` is a git command used for a lot of different things.
  - In the past it was used for switching branches, but newer versions of git use `git switch`
- `git checkout [branch-name]`
  or
  `git switch [branch-name]`
  switch to another branch
- `git checkout -b [new-branch-name]`
  or
  `git switch -c [new-branch-name]`
  create a new branch and switch to it at the same time

Merging and deleting branches

- (navigate to the main branch)
  `git merge [branch-name]`
  merge the specified branch into the current branch

- `git branch -d [branch-name]`
  delete a branch

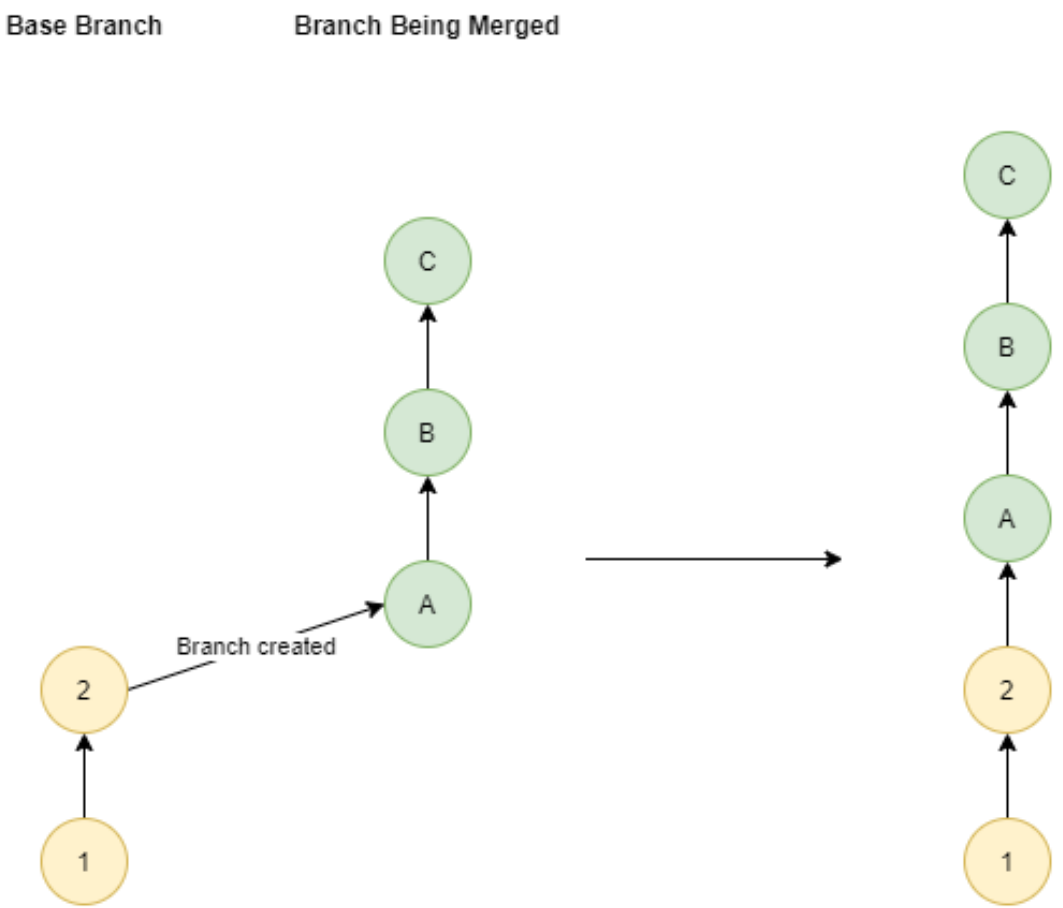
Common workflow

- Create new feature / fix a problem on a new branch
- Make changes
- Merge back into main
- Delete branch

Types of merges

1. Fast-forward

   - No other changes have been made to master (or current branch), so master is just “fast forwarded” to the point of the merged branch



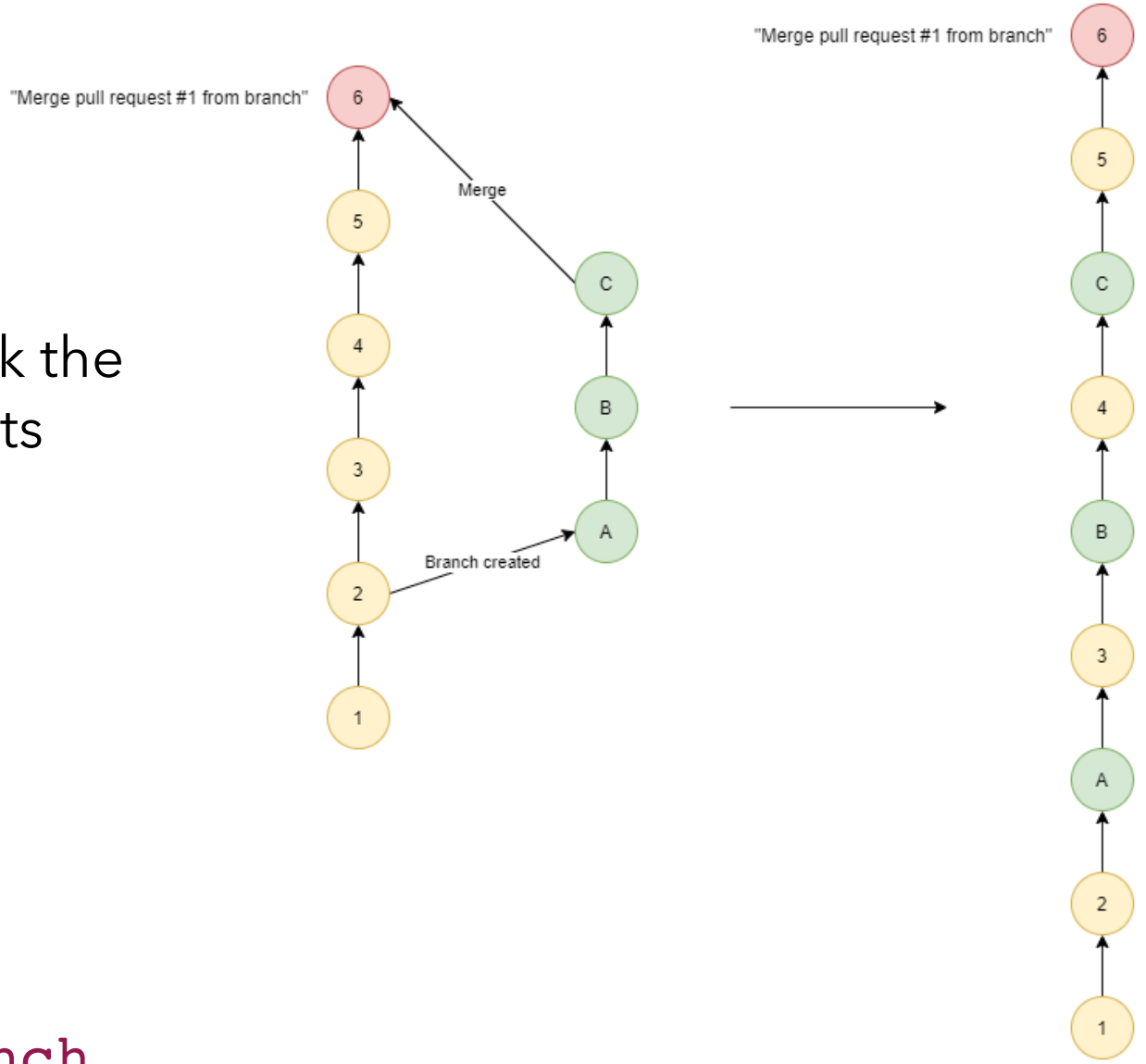
(while on main branch)

```
git merge new_branch
```

Types of merges

2. Merge commit

   - Commit history is preserved
   - Some people think the commit history gets messy



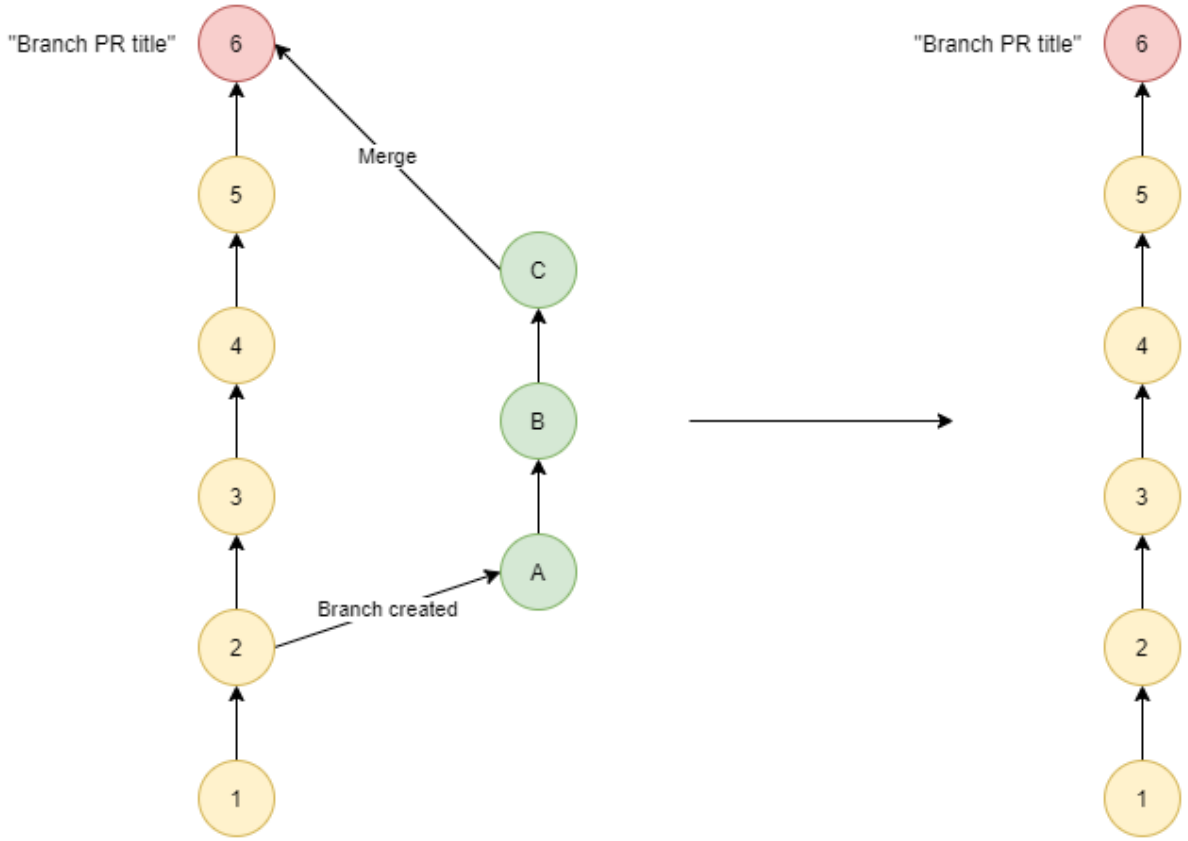
(while on main branch)

```
git merge new_branch
```

Types of merges

3. Squash & Merge

   - All commits on the branch are merged into one commit
   - Commit history is not preserved



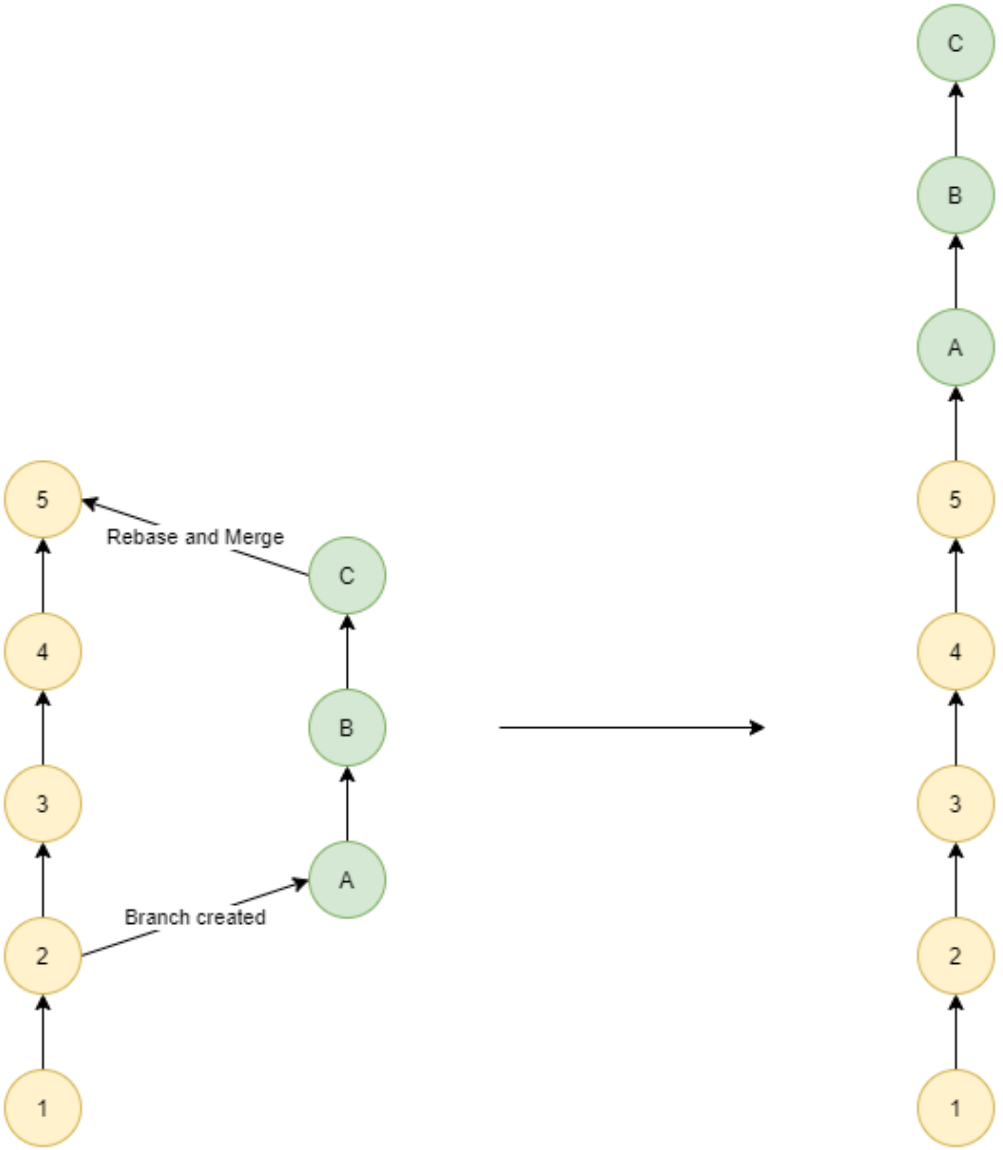
(while on main branch)

```
git merge --squash new_branch
```

Types of merges

4. Rebase

   - Stacks branch commits onto main commits
   - Sometimes rebasing is viewed as “changing history” and is discouraged



(while on new_branch branch)

```
git rebase main
```

Merge Conflicts

- Merge conflicts happen when you merge branches that have competing commits
  - Indicate that Git needs a human's help to decide which change to incorporate in the final merge
- See https://help.github.com/en/articles/about-merge-conflicts
- Merge conflicts are most common when collaborating with others

Tips to avoid merge conflicts

- Keep code lines short
- Keep commits small and focused
- Beware of stray edits to whitespace
- Merge often (if possible)
- Sync remote and local work **whenever** a change is made
- *Pull remote* repo into local before starting work
- *Fetch* remote repo and examine changes before *pushing*

View or return to past commits

Visiting past commits

- Git has a snapshot of the repository as it was at each commit
- We can easily view past commits:

  `git checkout XXXX` (using identifying hash in place of XXXX)

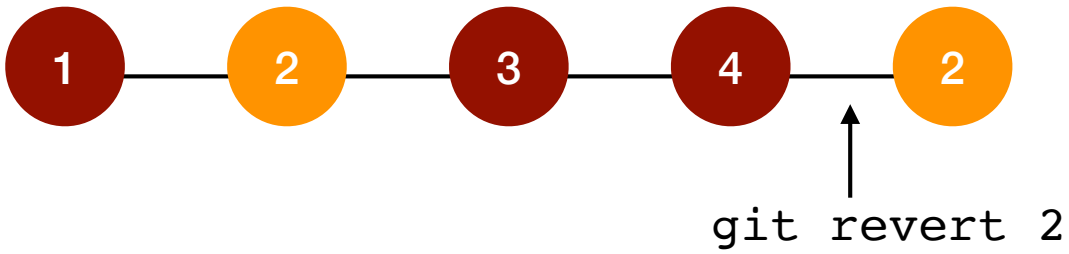
  - All current changes should be committed first
  - You can create a branch from the old commit if new changes are desired from this point

Return to past commits

- Suppose we want to revert to an old commit and continue our work from that point
- We can revert back to an old commit:

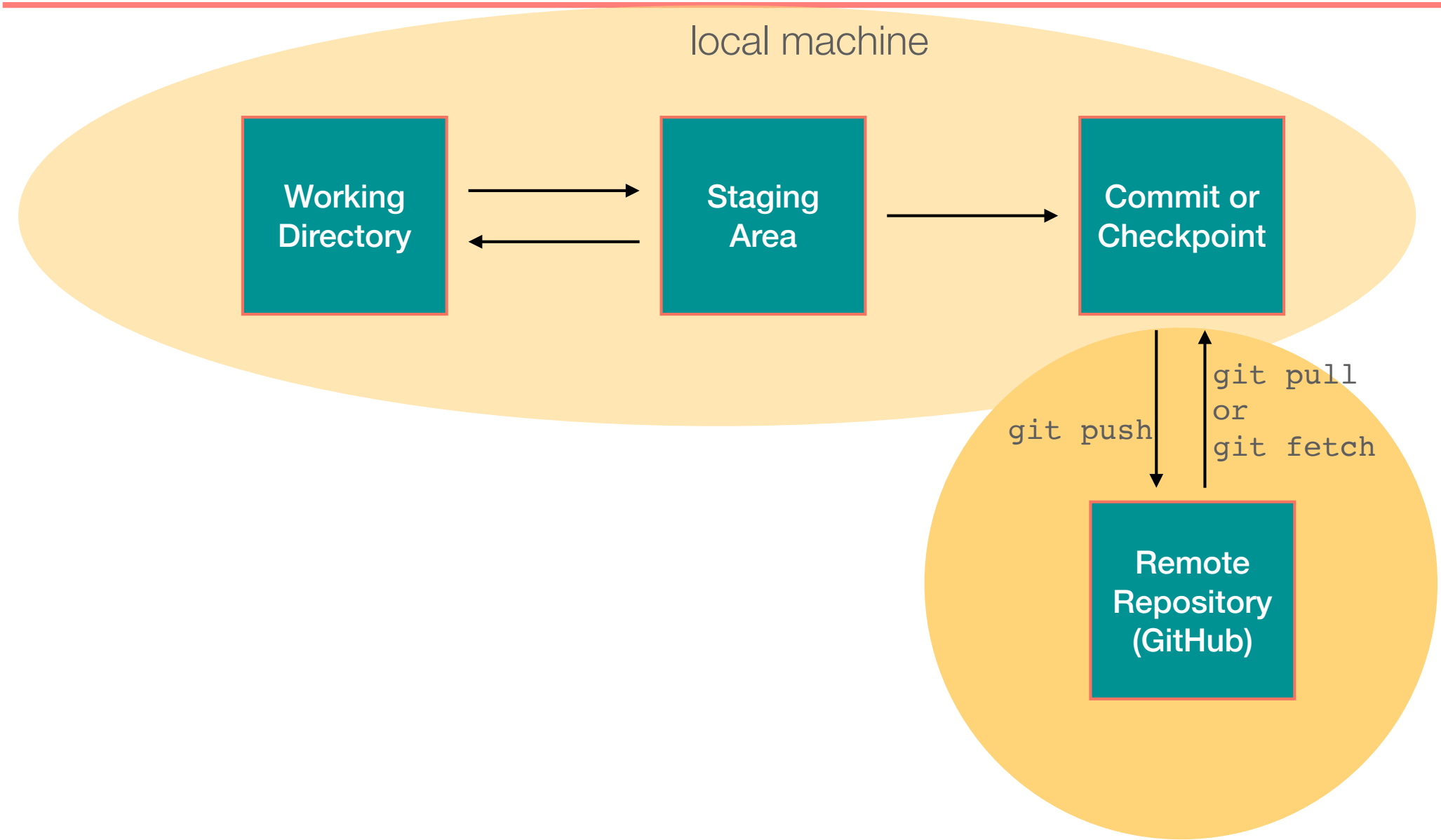
  `git revert XXXX` (using identifying hash in place of XXXX)

  - All commits will still be in commit history and the reverted version will now be the HEAD



remote repositories

Everything we've done so far has been local



Remote Options

- GitHub is just one cloud based option for remote repositories
- Many companies will have an in-house remote repository location
- Bitbucket
- Google Cloud Source Repositories

`git push`

- Pushing refers to sending your committed changes to a remote repository
- When you change something locally, you then **push** those changes to the remote repository, such as GitHub (so others can potentially see them)

Setting up remote

1. Create a blank repository in your GitHub account and copy the URL.

2. On your local machine, add the remote location:

   `git remote add origin <url>`

   for example:

   `git remote add origin https://github.com/user_name/repo_name.git`

3. Push the local repo to GitHub

   First time (this will tell GitHub make a remote branch called “main” that will be connected to the local branch “main”:

   `git push -u origin main`

   Subsequent times:

   `git push`

4. Refresh GitHub to see that it worked

Branch “origin/main”

- When we add the remote (with the alias origin), we are creating a remote branch called "origin/main".
- origin/main works like any other branch except that it can't be checked out
- You can see it with
  - `> git branch -r` (to see remote branches) or
  - `> git branch -a` (to see all branches)

Get changes from remote to local

```
> git pull
```

- Assumes that remote connection is already established

Tips

- Always pull before you start work on your local machine to ensure you have current version of repo
- It is good practice to pull before you push to see any changes that others have made

Working with remotes

Remote. *Working with remote (GitHub) repositories.*

`git remote -v`
show url for connected remote repo - displays nothing if remote is not set up

`git remote add [alias] [url]`
add url as remote alias - "origin" is a typical alias:
`git remote add origin https://github.com/user_name/repo_name.git`

`git push -u origin main`
push main branch to alias origin. the -u sets the upstream and should be used on first push

`git push`
push local changes to remote (2nd push and beyond)

`git pull`
pull remote changes into local repo

`git remote set-url origin [url]`
change the remote path

Cloning and Forking

Copy a remote repository

- Use the clone command to copy a remote repo (yours or any other public repo) onto your local machine
  - `git clone <url>`
- The remote repository path will automatically be set up to the cloned url
  - Ideal if you want to keep up to date with regular changes
  - If you don't have write permission, you won't be able to push any changes
  - Be careful of making local changes and pulling remote changes - the mixed commit history might cause problems
- To change the remote url:
  - `git remote set-url origin <new-url>`

Forking

- A fork is an independent copy of a repository
- Ideal for making personal changes to a repo
  - Pull requests can be used to request that your changes be merged into original repo
- There is no git command for forking
- Cloning and then changing remote url is similar to forking

Pull Requests

What is a Pull Request?

- Pull request are a GitHub* feature used for collaboration and code review
  - *Not unique to GitHub
- Pull requests are a way to introduce changes from one branch to another or from a fork to the original repository
- Pull requests allow developers to propose changes to another individual's repository

General Step of a Pull Request

1. **Fork** the repo you want to contribute to
2. Create a development **branch** in the forked repo to work on
3. **Commit** all changes to the development branch
4. **Submit** a formal pull request which asks the maintainer(s) of the original code to review your changes
5. Other developers, including repo maintainer(s), **review** the proposed changes
6. If changes are approved, the pull request is **merged** and the development branch is deleted
7. The pull request is **closed**

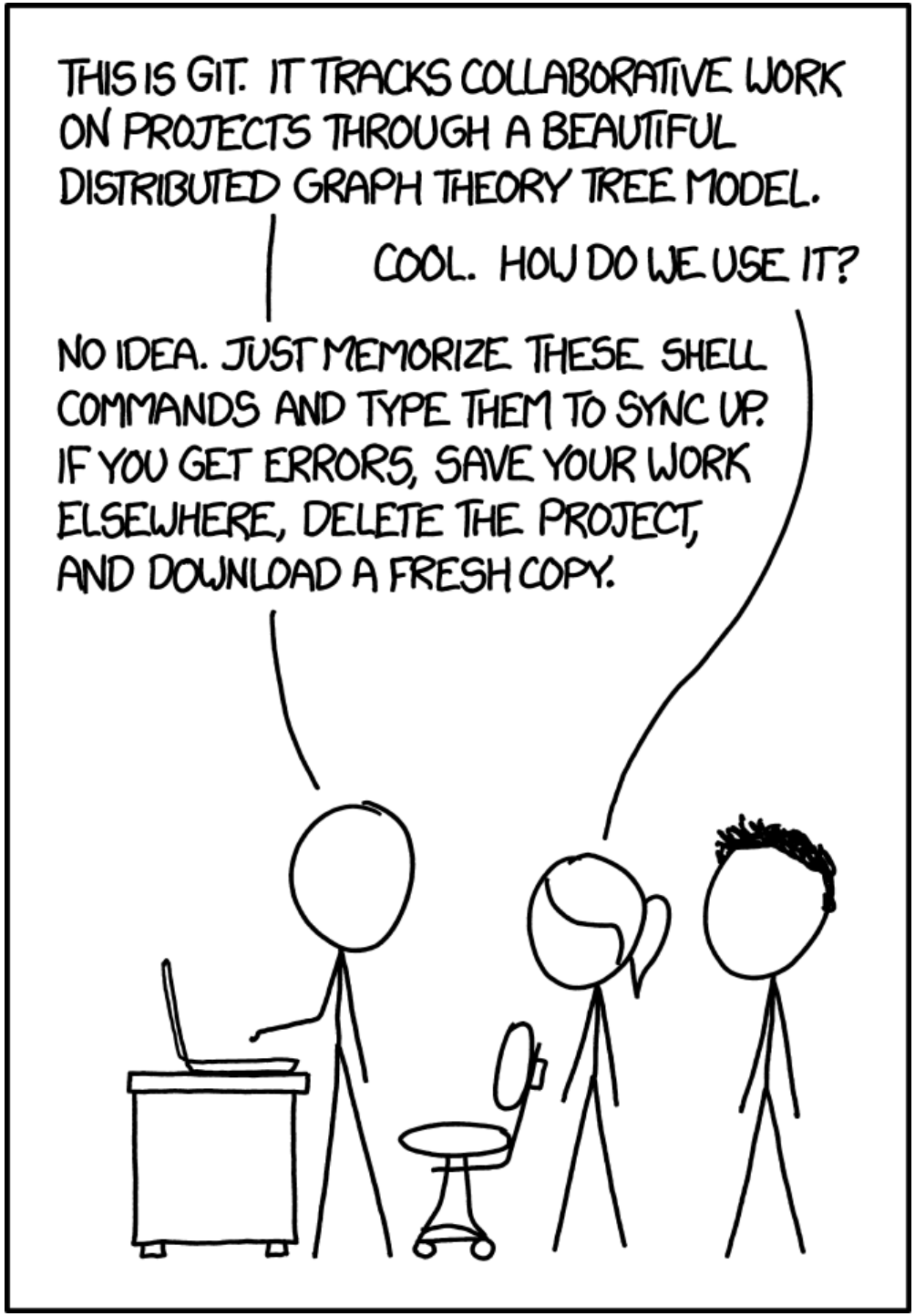
Keep practicing

Mastering Git

- Learning the git commands is relatively easy
- Living git takes time and practice
- There is a lot of functionality that we only briefly mention or haven't covered at all
- I'm still learning too!

Graphical Interfaces

- While we've been doing everything in the command line, there are many GUI options
- https://git-scm.com/downloads/guis/
- Many IDEs and/or editors manage Git including:
  - Rstudio
  - Atom
  - **Visual Studio**



https://xkcd.com/1597/

I must admit that I've resorted to this option in the past and sometimes instruct students to do so when I can't figure out your errors.

https://xkcd.com/1597/

Star Wars scrolling git log

http://starlogs.net/