Week 7

DATS 6450 – FOUNDATIONS OF COMPUTER SCIENCE

Twitter Accounts to Follow – People

@rasbt	@sirajraval
@andrewyng	@jeremyphoward
@wesmckinn	@smerity
@hadleywickham	@alsweigart
@dbader_org	@beeonaposy
@hugobowne	@benhamner
@rdpeng	@bork
@ronald_vanloon	@thomasp85
@chrisalbon	@juliasilge
@sentdex	@JennyBryan
@becomingdatasci	@justmarkham
@kirkdborne	@mdancho84
@dj44	@seb_ruder
@honnibal	@DynamicWebPaige
@_inesmontani	@karpathy
@math_rachel	@PyImageSearch
@aureliengeron	@dataandme

Twitter Accounts to Follow – Companies

@databricks	@mxlearn
@3blue1brown	@kaggle
@capitalonetech	@scikit_learn
@capitalonedevex	@facebookai
@ylecun	@r2d3us
@spacy_io	@stitchfix_algo
@codewisdom	@GoogleAI
@realpython	@explosion_ai
@deepmindai	@tryolabs
@pytorch	@DataCamp

What is version control?

A **version control system** is a tool that manages changes made to the files and directories in a project.

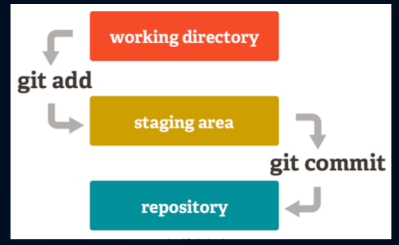
- Many version control systems exist
  - This lesson focuses on one called **Git**, which integrates with many of the data science tools covered in our other lessons.

- Version control systems keep track of every modification to your code in a special kind of database
  - If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

- Version control isn't just for software development
  - books, papers, parameter sets, and anything that changes over time or needs to be shared can and should be stored and shared using a version control system like Git.

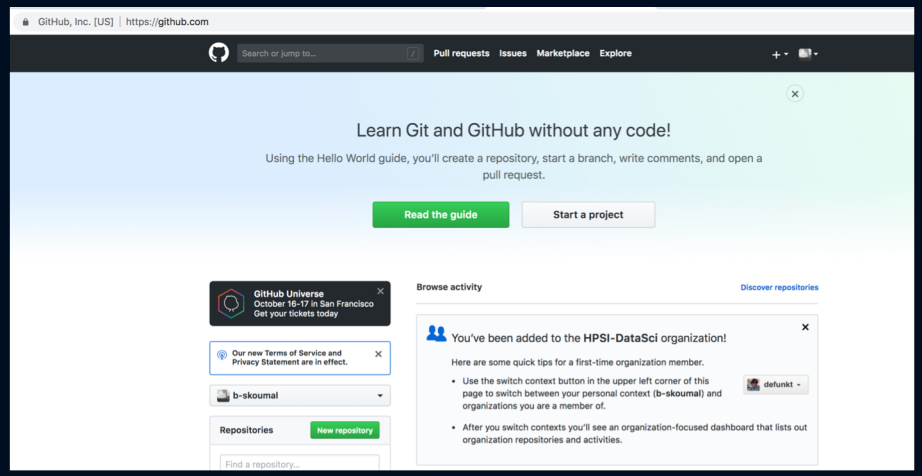
What is git/Github?

git is a version control system (VCS) - instead of storing a whole copy of every version of every file, it just stores the differences between each version.



There are other version control systems (subversion, mercurial, etc.)

GitHub is a remote repository hosting site, and social network built around git.



There are other repo hosting social networks that leverage git (BitBucket, GitLab, etc.).

Why version control?

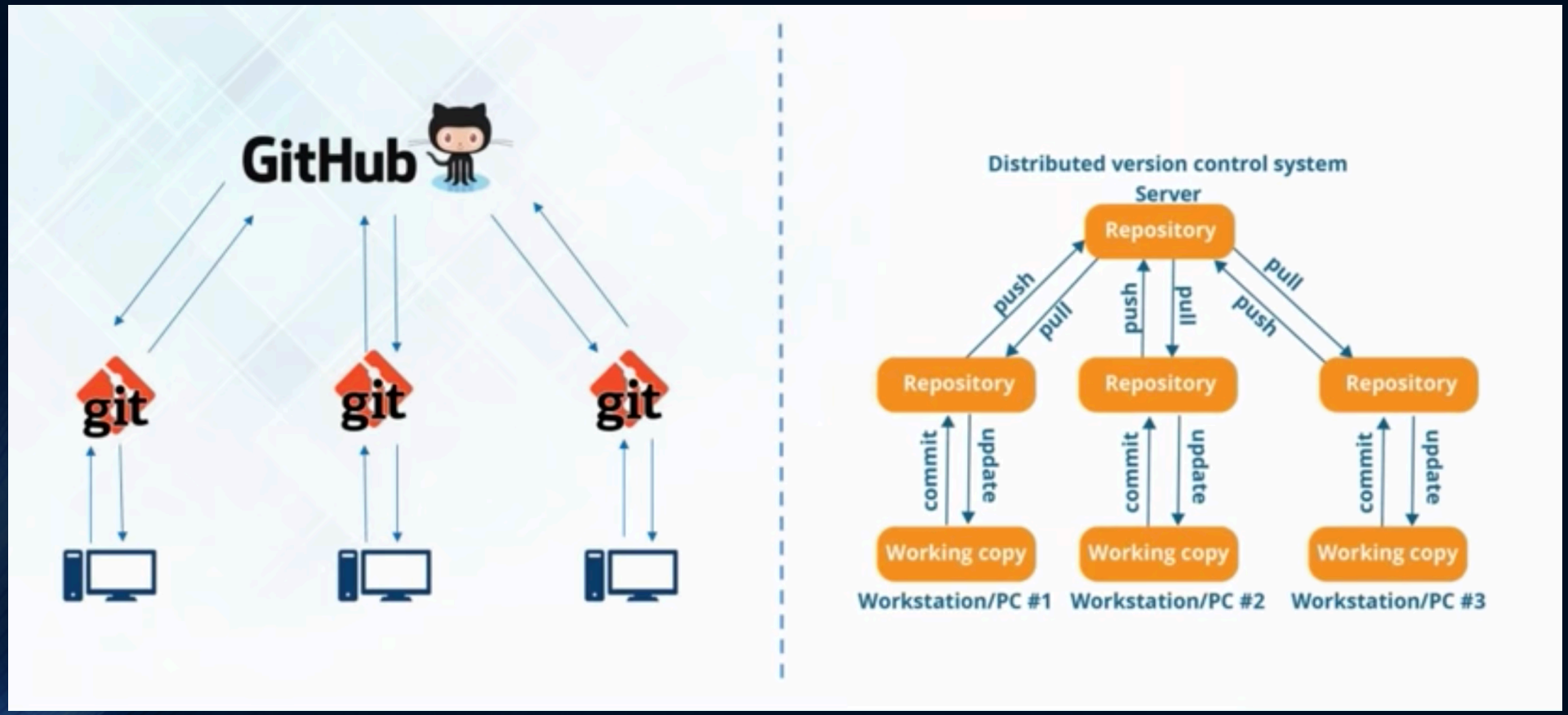
Strengths:

- Prevents idiocy - nothing that is saved to Git is ever lost, so you can always revert to a previous version of your code
- Git automatically notifies you when your work conflicts with someone else's, so it's harder (but not impossible) to accidentally overwrite work.
- Git can synchronize work done by different people on different machines, so it scales as your team does.

What is GitHub - Video

- https://www.youtube.com/watch?v=w3jLJU7DT5E

Why is it distributed?

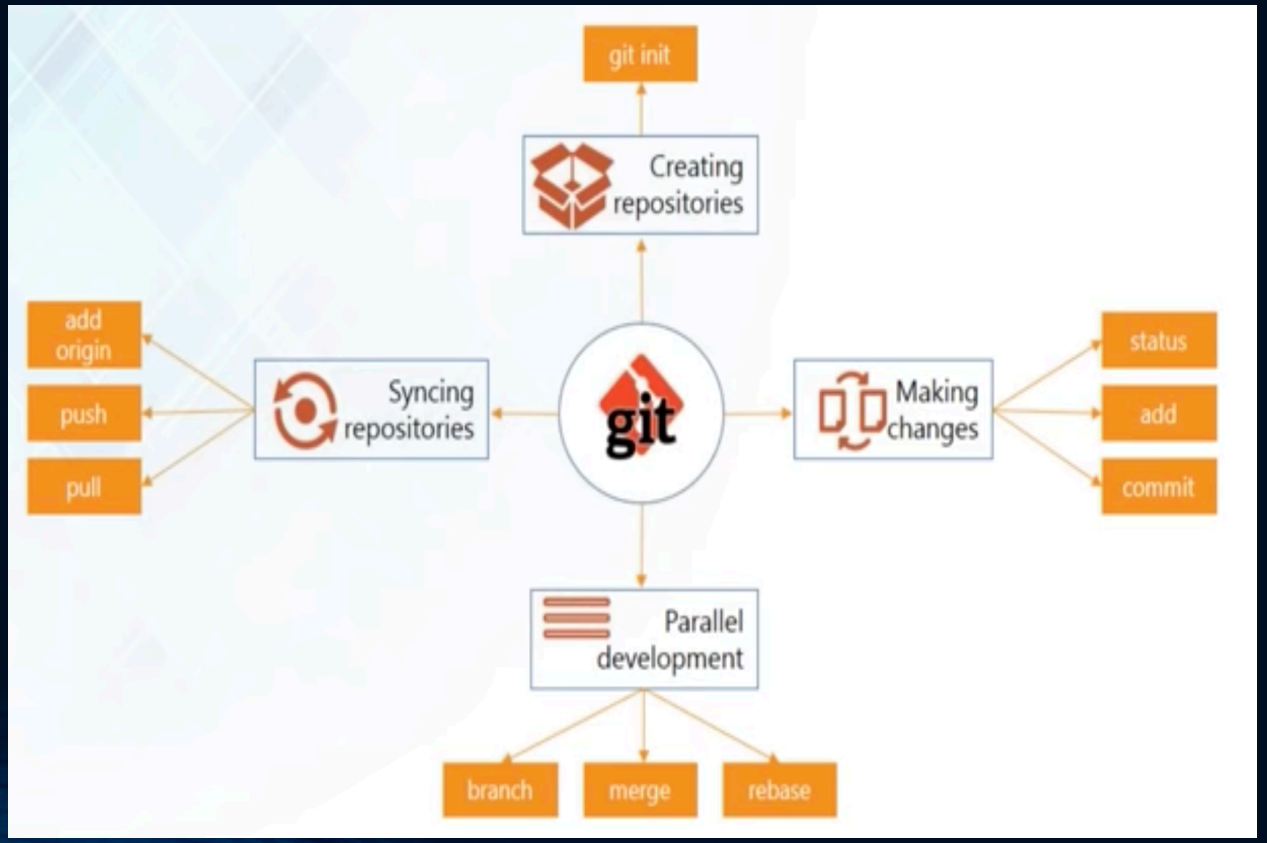


What we'll cover today

One of the criticisms for using git is that it has a relatively steep learning curve.

With that said, we hope to cover a few high-level points today:

- Initializing a new Git repo
- Cloning an existing Git repo
- Committing a modified version of a file to the repo
- Configuring a Git repo for remote collaboration



.git

Where does Git store information? What is a "repository"?

- Each of your Git projects has *two* parts:
  1. The files and directories that you create and edit directly
  2. The extra information that Git records about the project's history

- The combination of these two things is called a **repository**.

- Git stores all of its extra information in a directory called .git located in the root directory of the repository.

- Git expects this information to be laid out in a very precise way
  - We don't recommend editing/removing anything in your .git directory

git init

To "initialize" (create) a new repository, you'll use the command

`$ git init` - a one-time command you use during the initial setup of a new repo.

It can be used to convert an existing (local) project to a Git repository , or to simply initialize a new (empty) repository.

Executing git init creates a .git subdirectory in the current working directory, which contains all of the necessary git metadata for the new repository.

If you run `$ git init <directory>` the command is run inside of the folder you provide as the `<directory>` argument. If this directory does not exist, it will be created.

Note #1: most other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project

Note #2: If you've already run git init on a project directory and it contains a .git subdirectory, you can safely run git init again on the same project directory. It will not override an existing .git configuration.

Demo

CREATING A LOCAL GIT REPOSITORY

https://commons.wikimedia.org/wiki/File:Planets_are_us.png https://img00.deviantart.net/95f6/i/2009/156/f/8/werewolf_vs_dracula_by_b_maze.jpg
https://commons.wikimedia.org/wiki/File:Mummy_icon_-_Noun_Project_4070.svg https://commons.wikimedia.org/wiki/File:Lune_ico.png

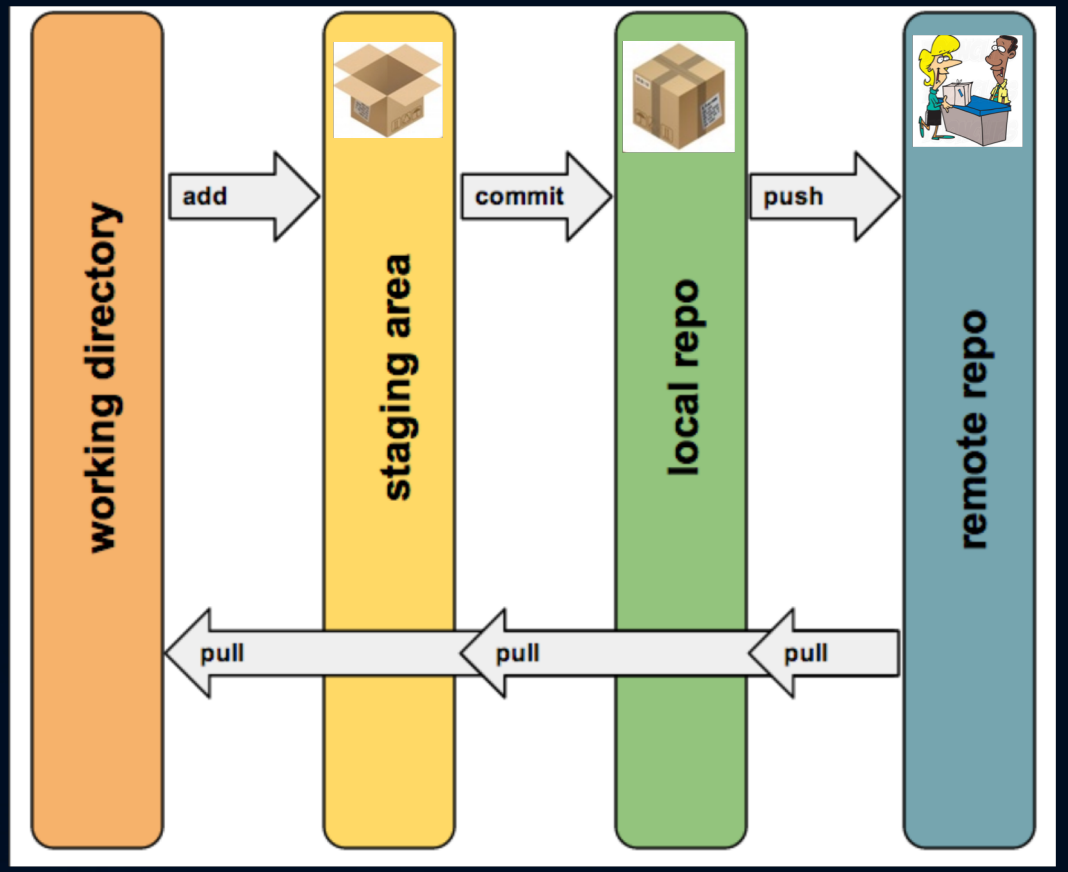
Different zones of git

staging area

Git has a staging area in which it stores files with changes you want to save that haven't been saved yet.

Putting files in the staging area is like putting things in a box, while committing those changes is like closing up that box, and pushing the box is like putting that box in the mail: you can add more things to the box or take things out as often as you want, but once you put it in the mail, you can't make further changes.

(github)



git status

How can I check the state of a repository?

When you are using git, you will frequently want to check the status of your repository. Assuming you are in an initialized git repo, if you run the command `$ git status`, it will display a list of the files that have been modified since the last time changes were saved.

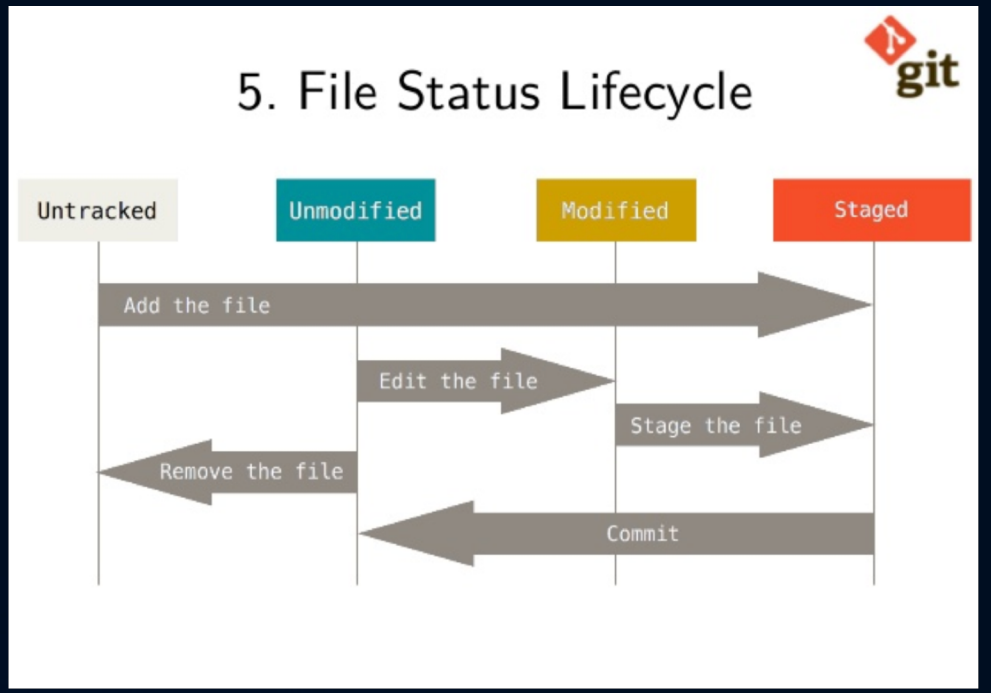
```
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
#         modified: hello.py
#
# Changes not staged for commit:
# (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working directory)
#
#         modified: main.py
#
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#
#         hello.pyc
```

git status

`$ git status` shows you which files are in this staging area, and which files have changes that haven't yet been put there.

Git has three states your files can be in at any moment in time.

1. **Modified:** You have a file you've made changes to *but* you have not yet committed (saved) to your local database.
2. **Staged**: You have a modified file that you've "marked" as one you are planning to include in your next commit "snapshot".
3. **Committed:** You "officially" saved all your changes/files to your local database in git.



git diff

How can I tell what I have changed?

In order to compare the file as it currently is to what you last saved, you can use `$ git diff filename`.

`$ git diff` without any filenames will show you all the changes in your repository

`$ git diff directory` will show you the changes to the files in some directory.

What is in a diff?

A diff is a formatted display of the differences between two sets of files.

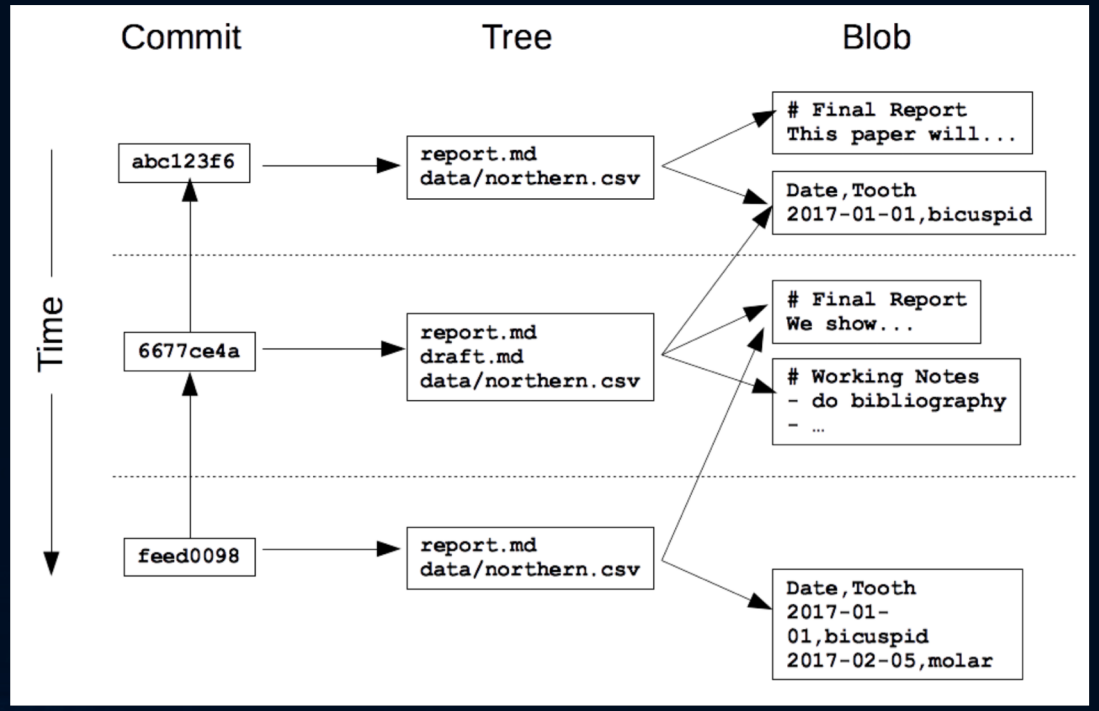
The `$ git diff` command is often used along with `$ git status` and `$ git log` to analyze the current state of a Git repo.

git commit

What is contained within a "commit"

Git uses a 3-level structure for storing commits:

1. A **commit** contains metadata such as the author, the commit message, and the time the commit happened.
   - The most recent commit is at the bottom (feed0098), and vertical arrows point up towards the previous ("parent") commits.
2. Each commit also has a **tree**, which tracks the names and locations in the repository when that commit happened.
   - In the oldest (top) commit, there were two files tracked by the repository.
3. For each of the files listed in the tree, there is a **blob**.
   - This contains a compressed snapshot of the contents of the file when the commit happened. (Blob is short for *binary large object*, which is a SQL database term for "may contain data of any kind".)
   - In the middle commit, report.md and draft.md were changed, so the blobs are shown next to that commit.
   - data/northern.csv didn't change in that commit, so the tree links to the blob from the previous commit.
   - Reusing blobs between commits helps make common operations fast and minimizes storage space.



Reference: Datacamp

commit

How do I commit changes?

To save the changes in the staging area, you use the command `$ git commit` . It always saves everything that is in the staging area as one unit: as you will see later, when you want to undo changes to a project, you undo all of a commit or none of it.

When you commit changes, git requires you to enter a log message. This serves the same purpose as a comment in a program: it tells the next person to examine the repository why you made a change.

By default, git launches a text editor to let you write this message. To keep things simple, you can use -m "some message in quotes" on the command line to enter a single-line message like this:

```
$ git commit –m ”adding logfile.txt”
```

If you accidentally mistype a commit message, you can change it using the --amend flag.

```
$ git commit --amend –m “new (fixed) message”
```

log

How can I view a repository's history?

The command `$ git log` is used to view the log of the project's history.

Log entries are shown most recent first, and look like this:

When you use `$ git commit –m “log message”`, the messages that you tag each commit with show up here (in the git log)

```
commit 043070548738119599 3bac9c21512ccfb511056d
Author: b-skoumal<brentskoumal@gwu.edu>
Date: Wed Apr 20 13:42:26 2018 +0000

          updated 'week7.pptx'

commit 87381054195993bac12ccfb519c215
Author: b-skoumal<brentskoumal@gwu.edu>
Date: Thur Apr 21 04:20:26 2018 +0000

          fixed mike’s annoying variable name

commit 42001054195993bac12ccfb59b4201
Author: b-skoumal<brentskoumal@gwu.edu>
Date: Thur Apr 21 04:20:26 2018 +0000

          added references to week7 folder
```

log

How do I write a better log message?

Writing a one-line log message with `$ git commit -m "message"` is good enough for very small changes, but your collaborators (including your future self) will appreciate more information.

If you run just `$ git commit` *(without* -m "message") git launches a text editor with a template like this →

```
# Please enter the commit message for your changes.
# Lines starting with '#' will be ignored, and an
# empty message aborts the commit.
#
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
# modified: brents_code.py
#
```

The lines starting with # are comments, and won't be saved. (They are there to remind you what you are supposed to do and what files you have changed.)

Your message should go at the top, and may be as long and as detailed as you want.

Demo

TRACKING CHANGES WITH GIT

hash

What is a hash?

Every commit to a repository has a unique identifier called a hash (since it is generated by running the changes through a random number generator called a hash function).

This hash is normally written as a 40-character hexadecimal string like 7c35a3ce607a14953f070f0f83b5d74c2296ef93, but most of the time, you only have to give git the first 6 or 8 characters in order to identify the commit you mean.

Hashes are what enable Git to share data efficiently between repositories. If two files are the same, their hashes are guaranteed to be the same. Similarly, if two commits contain the same files and have the same ancestors, their hashes will be the same as well. Git can therefore tell what information needs to be saved where by comparing hashes rather than comparing entire files.

HEAD

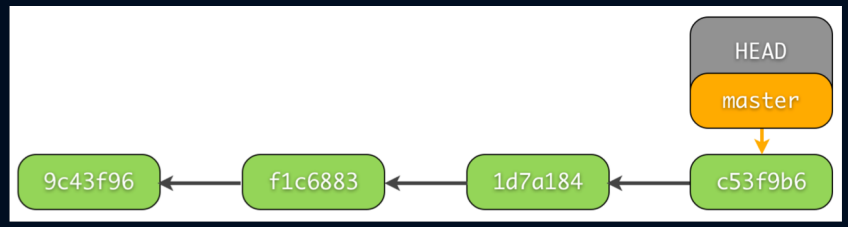
What is git's equivalent of a relative path?

A hash is like an absolute path: it identifies a specific commit. Another way to identify a commit is to use the equivalent of a relative path.

The special label `HEAD`, always refers to the most recent commit.

The label `HEAD~1` then refers to the commit before it, while `HEAD~2` refers to the commit before that, and so on.

Note that the symbol between `HEAD` and the number is a tilde `~`, *not* a minus sign `-`, and that there cannot be spaces before or after the tilde.



HEAD

How can I see what changed between two commits?

You can use the *show* command with a commit ID to show the changes made *in* a particular commit.

To see the changes *between* two commits, you can use `$ git diff ID1..ID2`, where ID1 and ID2 identify the two commits you're interested in, and the connector `..` is a pair of dots.

For example, `$ git diff abc123..def456` shows the differences between the commits abc123 and def456, while `$ git diff HEAD~1..HEAD~3` shows the differences between the state of the repository one commit in the past and its state three commits in the past.

Demo

EXPLORING HISTORY

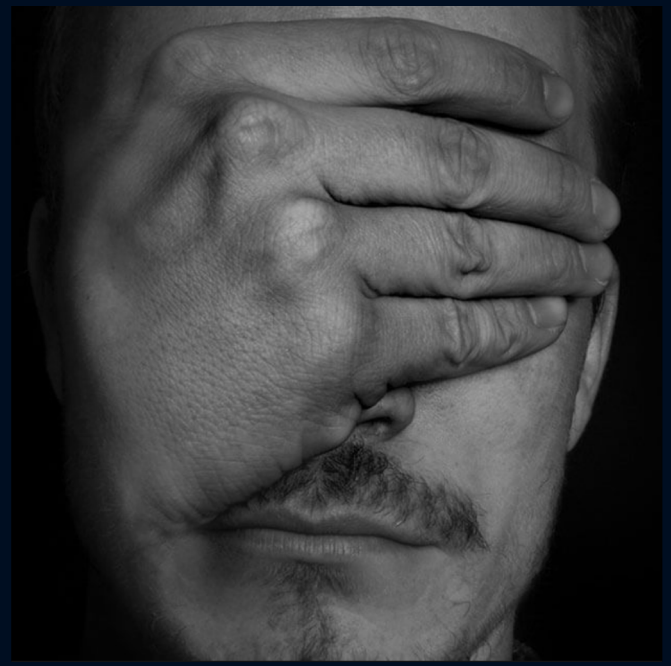
.gitignore

How do I tell Git to ignore certain files?

Data analysis often produces temporary or intermediate files that you don't want to save. (.pyc, .ipynb_checkpoints, error.log, etc.)

You can tell it to stop paying attention to (tracking changes on) files you don't care about by creating a file in the root directory of your repository called *.gitignore* and storing a list of wildcard patterns that specify the files you don't want git to pay attention to.

You can also simply list a directory, and git will ignore the directory itself (as well as all of its contents).



Demo

IGNORING THINGS

git clean

How can I remove unwanted files?

Git can help you clean up files that you have told it you don't want. The command `$ git clean -n` will show you a list of files that are in the repository, but whose history git is not currently tracking.

(this command basically shows you what files will be removed if you instead run `$ git clean -f`

Use this command carefully: git clean only works on untracked files, so by definition, their history has not been saved.

If you delete them with `$ git clean -f`, they're gone for good.

Recall, you can also use `$ git status` to see which files are untracked

git config

How can I see how Git is configured?

Like most complex pieces of software, git allows you to change its default settings.

To see what the settings are, you can use the command
`$ git config --list`

There are two optional flags you can also use,
`--global` : settings for every one of your projects.
`--local` : settings for one specific project.

local settings (per-project) take precedence over global settings (per-user)

`$ git config --list`

```
credential.helper=osxkeychain
user.name=brent_skoumal
user.email=brent.skoumal@gmail.com
credential.helper=osxkeychain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
core.ignorecase=true
core.precomposeunicode=true
submodule.active=.
remote.origin.url=https://github.com/danhtuan/deep_learning.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
```

`$ git config --list --local`

```
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
core.ignorecase=true
core.precomposeunicode=true
submodule.active=.
remote.origin.url=https://github.com/danhtuan/deep_learning.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
```

`$ git config --list --global`

```
user.name=brent_skoumal
user.email=brent.skoumal@gmail.com
credential.helper=osxkeychain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
```

git config

How can I change my Git configuration?

Most of git's settings should be left as they are. However, there are two you should set on every computer you use:

1. your name (user.name)
2. your email address (user.email)

These are recorded in the log every time you commit a change, and are often used to identify the authors of a project's content in order to give credit (or assign blame, depending on the circumstances).

To change a configuration value for all of your projects on a particular computer (which is what most students in DATS6450 will be doing), run the following command:

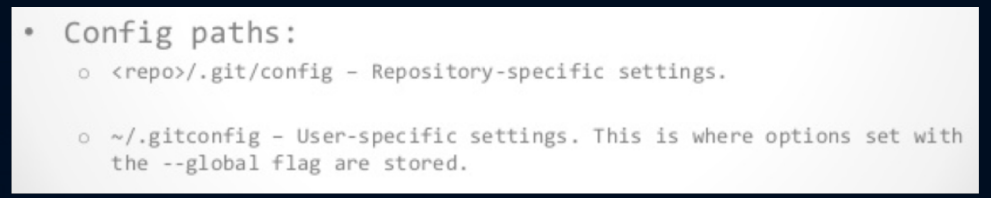
```
$ git config --global setting.name setting.value
```

For example:
To set your user.name:

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

To set your user.email:

```
$ git config --global user.email brent.skoumal@gmail.com
```



The Test

- Test will be on weeks 1-6
  - Git will not be on the test
- Format:
  - Multiple choice
  - Fill-in-the-blank
  - Write the command