Backlinks (1)

Git

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 is used by many of the data science tools covered in our other lessons. Its strengths are:

- **Keep track of changes to files**: Nothing that is saved to Git is ever lost, so you can always go back to see which results were generated by which versions of your programs.
- **Notice conflicts between changes made by different people**: Git automatically notifies you when your work conflicts with someone else's, so it's harder (but not impossible) to accidentally overwrite work.
- **Synchronize files between different computers**: Git can synchronize work done by different people on different machines, so it scales as your team does.

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

Where does Git store information?

Each of your Git projects has two parts:

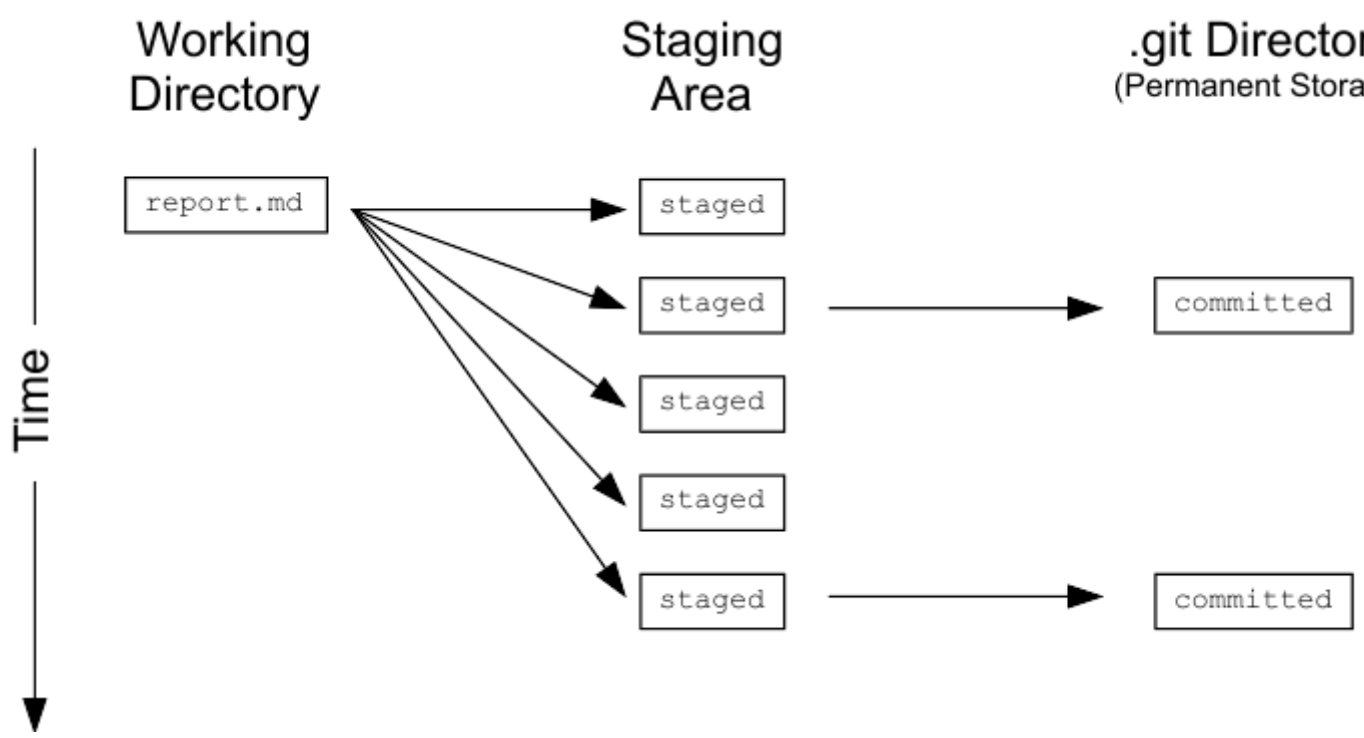
1. the files and directories that you create and edit directly, and
2. the extra information that Git records about the project's history.

- 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, so you should never edit or delete anything in `.git`.

Suppose your home directory `/home/repl` contains a repository called `dental`, which has a sub-directory called `data`. Where is information about the history of the files in `/home/repl/dental/data` stored?
Answer:
/home/repl/dental/.git

How can I tell what I have changed?

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 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.



Desktop programming tools like RStudio can turn diffs like this into a more readable side-by-side display of changes; you can also use standalone tools like DiffMerge or WinMerge.

How can I edit a file?

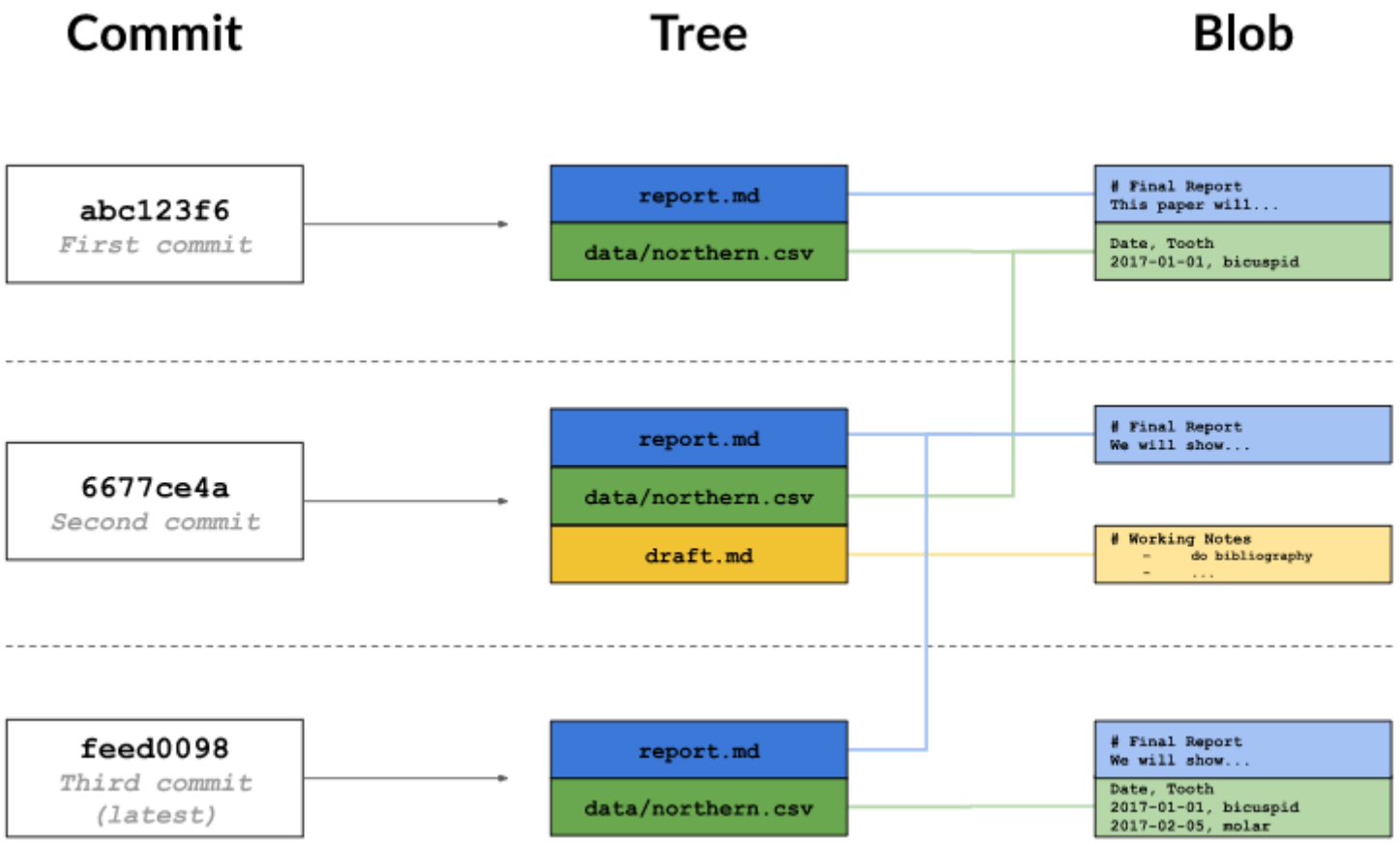
- one Unix text editor s called Nano
- if you type `nano filename`, it will open `filename` for editing (or create it if it doesn't already exist). You can then move around with the arrow keys, delete characters with the backspace key, and so on. You can also do a few other operations with control-key combinations:

- Ctrl-K: delete a line.
- Ctrl-U: un-delete a line.
- Ctrl-O: save the file ('O' stands for 'output').
- Ctrl-X: exit the editor.

How does Git store information?

Git uses a three-level structure for this.

1. A **commit** contains metadata such as the author, the commit message, and the time the commit happened. In the diagram below, the most recent commit is at the bottom (`feed0098`), underneath its 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 help make common operations fast and minimizes storage space.



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 pseudo-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.
- enable Git to share data efficiently between repositories.
- if two commits contain the **same files and have the same ancestors,** their hashes will be the same as well
- what information needs to be saved where by comparing hashes rather than comparing entire files
- hash is like an absolute path: it identifies a specific commit

Basic workflow	
Check state of a repository • displays a list of the files that have been modified since the last time changes were saved. • hows you which files are in this staging area, and which files have changes that haven't yet been put there	cd dental git status
Compare the file/directory/all changes in directory (without filenames) as it currently is to what you last saved • you can compare the state of your files with those in the staging area, you can use . The `-r` flag means "compare to a particular revision", and `HEAD` is a shortcut meaning "the most recent commit".	git diff filename/directory git diff -r HEAD path/to/file

• You can restrict the results to a single file or directory	
Saving changes 1. add file to staging area 2. commit everything in staging area • log message required to commit changes • can ammend • Git does not track files by default, it waits until you have used `git add` at least once • the untracked files won't have a blob, and won't be listed in a tree, i.e. won't benefit from version control • use git status to check files that are in your repository but aren't (yet) being tracked. • good practice to save periodically	cd dental (git status) git add report.txt git commit -m "Adding a reference" git commit --amend - m "new message"
Edit a file • when run git commit without message, git launches a text editor	nano names.txt (type info) (Ctrl-O to write the file out and save what is written) (Ctrl-X and Enter to exit the editor)
Repositories	
View a repository's history • Log entries are shown most recent first • The commit line displays a unique ID for the commit called a hash	git log (path) (Press the space bar to go down a page or the 'q' key to quit.)
View a specific commit • git log + git diff	git show 0da2f7 (q to quit) for relative path - commit made just before the most recent one: git show HEAD~1
See who changed what in a file • Each line contains five elements, 1. The first eight digits of the hash, `04307054`. 2. The author, `Rep Loop`. 3. The time of the commit, `2017-09-20 13:42:26 +0000`. 4. The line number, `1`. 5. The contents of the line, `# Seasonal Dental Surgeries (2017) 2017-18`.	git annotate report.txt
See what changed between two commits	git diff ID1..ID2 git diff HEAD~1..HEAD~3

Tell Git to ignore certain files
• for temporary or intermediate files that you don't want to save
	• 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
	• to ignore any file or directory called `build` (and, if it's a directory, anything in it), as well as any file whose name ends in `.mpl`. THEN `.gitignore` should contain:
	`build`
	`*.mpl`
	`pdf (should be *.pdf)`
	`*.pyc  -> ignore bin/analyze.pyc`
	`backup -> ignore backup/northern.csv`

Remove unwanted (untracked) files • git clean -n will show you a list of files that are in the repository, but whose history Git is not	cd dental git status git clean -f

currently tracking • git clean -f will then delete those files. • `git clean` only works on untracked files • ls to list the files in your current working directory. backup.log should no longer be there!	>removing backup.log ls >bin data report.txt results
See how Git is configured `git config --list` with one of three additional options: • `--system` : settings for every user on this computer. • `--global` : settings for every one of your projects. • `--local` : settings for one specific project. Each level overrides the one above it, so **local settings** (per-project) take precedence over **global settings** (per-user), which in turn take precedence over **system settings** (for all users on the computer).	git config --list --local
Change my Git configuration • there are two you should set on every computer you use: your name and your email address • sed to identify the authors of a project's content in order to give credit (or assign blame, depending on the circumstances).	git config --global user.email rep.loop@datacamp.com To change the email address (user.email) configured for the current user for all projects to rep.loop@datacamp.com.
Undo	
Unstage a file you shouldn't have	git reset HEAD
Undo changes to unstaged files • `--` must be there to separate • once you discard changes in this way, they are gone forever.	git checkout -- data/northern.csv
Undo changes to staged files • step 1: unstage the file • step 2: undo changes since the last commit	git reset HEAD path/to/file git checkout -- path/to/file
Restore an old version of a file • restore versions of that file from a commit • committing as saving your work, **checking out** as loading that saved version • takes two arguments: the hash that identifies the version you want to restore, and the name of the file. • Restoring a file doesn't erase any of the repository's history. Instead, the act of restoring the file is saved as another commit, because you might later want to undo your undoing. • Passing `-` followed by a number restricts the output to that many commits. • cat print the content of a file onto the standard output stream	cat data/western.csv git log -2 data/western.csv git checkout 2242bd data/western.csv cat data/western.csv (to display the updated contents) git commit -m "restored"
Undo all of the changes I have made • give `git reset` a directory. For example, `git reset HEAD data` will unstage any files from the `data` directory • if you don't provide any files or directories, it will unstage everything • `HEAD` is the default commit to unstage, so you can simply write `git reset` to unstage everything	git reset To remove all files from the staging area git checkout -- . To put those files back in their previous state

• `git checkout -- data` will then restore the files in the `data` directory to their previous state. • So `git checkout -- .` will revert all files in the current directory.	

What is a branch?

- allows you to have multiple versions of your work, and lets you track each version systematically.
- instead of creating different subdirectories to hold different versions of your project in different states which could cause confusion on which is the right version of each file in the right subdirectory, and you risk losing work.
- one branch do not affect other branches (until you **merge** them back together).
- Branches are the reason Git needs both trees and commits: a **commit will have two parents** when **branches are being merged.**
  - *blobs* for files, *trees* for the **saved states of the repositories**, and *commits* to record the changes
- By default, every Git repository has a branch called `master`

What are conflicts?

- for example, bug fixes might touch the same lines of code, or analyses in two different branches may both append new (and different) records to a summary data file.
- In this case, Git relies on you to reconcile the conflicting changes.
- Example
  - file `todo.txt`

```
A) Write report.
B) Submit report.
```

  - create a branch called `update` and modify the file to be:

```
A) Write report.
B) Submit final version.
C) Submit expenses.
```

  - switch back to the `master` branch and delete the first line, so that the file contains:

```
B) Submit report.
```

When you try to merge `update` and `master`, what conflicts does Git report?
-B) Submit report.
+A) Write report.
+B) Submit final version.
+C) Submit expenses.

  - Git can merge the deletion of line A and the addition of line C automatically.

How can I find out where a cloned repository originated?

- Git remembers where the original repository was. It does this by storing a **remote** in the new repository's configuration. A remote is like a browser bookmark with a name and a URL.
- If you use an online git repository hosting service like GitHub or Bitbucket, a common task would be that you clone a repository from that site to work locally on your computer. **Then the copy on the website is the remote.**

Pull and Push changes from a remote repository?

- remote repository is often a repository in an online hosting service like GitHub.
- A typical workflow is that you pull in your collaborators' work from the remote repository so you have the latest version of everything, do some work yourself, then push your work back to the remote so that your collaborators have access to it.

Branch	
List all of the branches in a repository • The branch you are currently in will be shown with a `*` beside its name.	git branch
View the differences between branches • same as revisions • to check what conflict Git report before merging	git diff branch-1..branch-2

Switch from one branch to another + Remove file • same as restoring version using hash • Git will only let you do this if all of your changes have been committed - can get around this • `git rm` removes the file then stages the removal of that file with `git add`, all in one step.	cd dental git branch #switch to summary-statistics branch git checkout summary-statistics #delete report.txt git rm report.txt git commit -m "Removing report" #check that it's gone ls #switch back to master branch git checkout master ls -report.txt in master branch should not be affected
Create a branch • using -b flag allows you to create a branch then switch to it in one step • The contents of the new branch are initially identical to the contents of the original. Once you start making changes, they only affect the new branch.	cd dental git branch git checkout -b deleting-report git rm report.txt git commit -m "report deleted" #to compare the master branch with the new state of the deleting-report branch git diff master..deleting-report
Merge two branches • When you merge source branch into destination source, Git incorporates the changes made to the source branch into the destination branch • If those changes don't overlap, the result is a new commit in the destination branch that includes everything from the source branch • Git automatically opens an editor so that you can write a log message for the merge	cd dental git merge source destination Ctrl+O and then Ctrl+X to exit this. If you want to avoid this, use the `--no-edit` flag in the `git merge` command.
Merge two branches with conflicts • running `git status` after the merge reminds you which files have conflicts that you need to resolve by printing `both modified:` beside the files' names. • In many cases, the destination branch name will be `HEAD` because you will be merging into the current branch. • To resolve the conflict, edit the file to remove the markers and make whatever other changes are needed to reconcile the changes, then commit those changes.	cd dental git merge alter-report-title master #to check which file has conflict git status #edit text file to reconcile changes nano report.txt Ctrl + K to remove entire lines Ctrl + O to Save then enter Ctrl + X to exit git add report.txt git commit -m "remove lines"
Repository	
Create a brand new repository • One thing you should *not* do is create one Git repository inside another. • Nested repositories may be used in very large projects but no ideal	git init project-name
Turn an existing project into a Git repository • after initializing the folder into a repo, Git immediately notices that there are a bunch of changes that can be staged (and committed afterwards)	git init (in the project's root directory) or `git init /path/to/project`
Example - in directory dental with is not yet a Git repository	pwd #'present working directory' git init git status

	git status

Create a copy of an existing repository
• When? Join a project that is already running, inherit a project from someone else, or continue working on one of your own projects on a new machine
• When you clone a repository, Git uses the name of the existing repository as the name of the clone's root directory, for example:
`git clone /existing/project`
will create a new directory called `project` inside your home directory.
`git clone /existing/project newprojectname`
to add new name
	pwd
	ls
	git clone /home/thunk/repo dental
	git clone
	`https://github.com/datacamp/project.git`

List names of remotes (cloned repos) • -v flag shows the remote's URLs.	git remote -v (for "verbose")
Define remotes • When you clone a repository, Git automatically creates a remote called `origin` that points to the original repository. • You can connect any two Git repositories this way, but in practice, you will almost always connect repositories that share some common ancestry.	
Add more remotes	`git remote add remote-name URL` `git remote add thunk /home/thunk/repo`
Remove existing remotes	`git remote rm remote-name`

Pull in changes from a remote repository
• **pull** changes from those repositories and **push** changes to them
• gets everything in `branch` in the remote repository identified by `remote` and merges it into the current branch of your local repository
• get changes from `latest-analysis` branch in the repository associated with the remote called `thunk` and merge them into your `quarterly-report` branch.
	git pull remote branch
	`cd dental`
	`git pull thunk latest-analysis`

Pull with unsaved changes • Just as Git stops you from switching branches when you have unsaved work • pulling in changes from a remote repository when doing so might overwrite things you have done locally • Fix: either **commit your local changes** or **revert them**, and then try to pull again. • commit all your local changes if you want your git pull to run smoothly	cd dental git pull origin master > overwrite warning #to discard changes in the repo git checkout -- report.txt git pull origin master
Push changes to a remote repository • push local changes - contents of your branch `branch-name` into a branch with the same name in the remote repository associated with `remote-name` • not good practice to use different branch names	cd dental git add data/northern.csv git commit -m "Added more northern data" git push origin master
Push with unsaved changes	git push origin master

Push with unsaved changes • Git does not allow you to push changes to a remote repository unless you have merged the contents of the remote repository into your own work. • Fix: bring your repository up to date with origin with git pull	git push origin master >error: failed to push some refsto '/home/thunk/repo' hint: Updates were rejected because the tip of your **current branch is behind** hint: its remote counterpart. Integrate the remote changes (e.g.hint: 'git pull ...') before pushing again. hint: See the 'Note about fast-forwards' in 'git push --help' for details. Ctrl + X git push origin master