Version control with git (for economists)

Luís Fonseca

London Business School

BPLIM Workshop on Reproducible Research and Modern Data Analysis

16 December 2019

Outline

1. What version control is
2. Using git for version control

Outline

1. What version control is

Version control and reproducibility

- Many journals now require data & code with submissions
- Economists work with code; but it seems we are not adopting many important tools
- Proper version control makes reproducing your results easier
- Co-authored papers are the most frequent. Version control allows easier tracking of collaboration and changes
- Useful even in solo projects

What are (most of us) we doing now?

Example folder from Gentzkow and Shapiro: [1]

cleandata_022113.do	cleandata_022613.do	regressions.log
cleandata_022113a.do	cleandata_022613_jms.do	regressions_022413.do
chips.csv	tvdata.dta	regressions_022713_mg.do
regressions_022413.log		

- What is correct here? Multiple versions for the same file
  - allows easy roll-back of changes
  - facilitates comparisons
- What is wrong?
  - Need to tag different dates and ownership in file names
  - File names may refer to a date while last change is on another
  - Why was the new file created?
  - Testing small changes with dependencies is messier
- No programmer does it like this
- This can be done much better and more systematically

[1] *Code and Data for the Social Sciences: A Practitioner's Guide*

What does a good version control system do?

- Backs up your project at every point since its beginning
- Synchronizes files with co-authors and machines
- Short & long-term undo: restore one file or entire project easily
- Tracks changes to any file and stores comments
- Tracks who did every change (there's even `git blame`!)
- Branching and merging: experiment safely; integrate when done

Most popular version control tools are:

- distributed: git and mercurial
- centralized: subversion

Why is Dropbox not good enough?

Dropbox provides very reliable cloud backups + ability to restore versions at least a month old (more if you pay for extended history)

It is a bad version control tool. Key differences:

1. git can restore a project easily to any point since its beginning; Dropbox can restore back to a short window of time
2. git keeps versions of projects. Dropbox keeps versions of files; to restore project to a date, need to go file by file, including deleted
3. Dropbox relies on the cloud to restore files. git works locally
4. git easily compares all changes between two versions of a project
5. git tells you which lines of code have conflicts when there are simultaneous changes; Dropbox just stores the two file versions
6. git tracks comments with changes; Dropbox only keeps a number
7. git allows branching, which is great (to be seen later)

Outline

git vs GitHub

- git is the actual software that helps you do version control
- GitHub is just the most popular platform that hosts git repos
- They are completely independent of each other
- There are other similar platforms: GitLab, Bitbucket
- GitHub is used here; for today's purposes, they're indistinguishable, except visually
- All offer free unlimited private repos
- Hosting on your own server:
  - Free, open source: GitLab CE, Gitea, Gogs
  - Paid, proprietary: GitHub, GitLab EE, BitBucket

Why is git so popular?

- It's fast to store changes and restore old versions
- Branching and merging work well
- Has tools to help debugging
- Works locally and decentralized
- Created by Linus Torvalds of Linux in 2005. Open source, free to use

Glossary (simplified)

- **Repo(sitory)**: git database that stores the files & the project history
- **Local**: your computer or the repo stored in your computer
- **Remote**: the server or the repo stored in the server to which you and your collaborators push changes
- **Working tree/directory/copy**: version of project visible in local folder
- **Commit**: (saves) a snapshot of your project at a given point in time; identified by its hash
- **Hash**: string like `24b9da6552252987aa493b52f8696cd6d3b00373` that identifies a commit; can be referred by first few characters
- **Push**: upload latest commit and its history to remote server (e.g. GitHub)
- **Fetch**: download changes from remote server but do not merge them
- **Merge**: combine two commits (snapshots) of a repo
- **Pull**: fetch+merge. Download & update local repo with changes stored remotely
- **Stage**: add to the staging area / index
- **Staging area / Index**: changes to be included in next commit
- **Clone**: download the entire history of a repo to your computer
- **Branch**: path in the history of repo. Main branch usually called `master`; others can be created (e.g. `develop`, `fix_bug_X`)
- **Checkout**: update working tree to reflect a specific commit or branch
- **Head**: the currently checked out commit

Best reference material

- Pro Git book freely available at https://git-scm.com/book/en/v2
  - Chapters 1 to 3 are enough
- Git documentation available at https://git-scm.com/docs
  - More confusing if not used to structure of the manual pages

GUI vs Command line

- Command line can be more intimidating initially
- Useful to learn how git works and what are interfaces doing
- I recommend starting with command line; being comfortable makes everything possible
- Can move to a GUI later: Sourcetree is my favorite on Windows
- Other common ones: Sublime Merge, GitKraken

Where to keep repos? What about backups?

Tastes vary. My workflow:

- Repos in a normal computer folder
- I don't mix repos and Dropbox. There are "hacks" online, but consensus says it's risky
- Dropbox to store large datasets, large unchanging documents and outputs
- GitHub to store my repos privately in the cloud
- How are repos backed up?
    - On a daily basis:
        - Laptop has a copy of the repo up to the last time I pulled
        - The school desktop has another
        - GitHub has a third copy
    - For a more systematic backup:
        - A script clones all my GitHub repos to a zip file and copies it to a dropbox folder.
        - I run this regularly

What to store in your git repo

- git is optimized for text (code) files
- git can also store other types of files (called binaries) such as pdfs, images, stata .dta files, etc. However:
    - it cannot compare different versions of these
    - storage is much less efficient:
        - text files: git can compare files and store only the changes
        - binaries: git stores all versions of file, even if changes are small
- Typical workflow: store only source code and source data
    - Outputs should be reproducible from source code and data
    - git will ignore folders or file types listed in a `.gitignore` file
    - Tools like Make (or project for Stata or drake for R) can be used to automate recreation of output from original source files

Installing git

- Follow instructions in https://git-scm.com/book/en/v2/Getting-Started-Installing-Git and https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup
- When installing, you will be asked to specify a text editor for when git requires you to write commit messages or to look at conflicting changes
- After installing, open a command prompt/terminal/bash window
- Here I used Git Bash in Windows (right-click in a folder, then "Git Bash here" to open terminal window)
- Just run `git --version` in bash to ensure it's installed.
- The first time you connect with GitHub or another remote host, you will need to login
  - These can be saved. See GitHub's help pages or Google it if it keeps asking you to login

The history of a repo

- We'll (try to) learn the key concepts of git.
- Understanding the model is more important than going through all the commands
- You'll figure out what you need as you go along
- Many people use git; any questions are easily googleable
- I will skip more advanced topics; goal is to leave you with a solid understanding of the basics

Note:
In the following slides, the gray blocks represent a terminal window where I'm running commands.

The lines starting with $ and in blue font are commands I execute.

The black font lines are the output from the terminal to my commands.

The history of a repo

You can create a repo(sitory) in any folder with `git init`.

```
$ git init
Initialized empty Git repository in
    (...)/git-tutorial/.git/
```

Notice this creates a hidden `.git` folder that stores everything. You can take a look, but don't do anything inside. All the information about your repo is inside that folder: the content (stored compressed), the history, changes, messages, authors.

If you plan to keep the repo in GitHub, it's easier to first create an empty repo there and then clone it locally using the browser address of the repo in GitHub. This will automatically configure the link to the remote server in GitHub.

```
$ git clone https://github.com/USERNAME/git-tutorial.git
Cloning into 'git-tutorial'...
warning: You appear to have cloned an empty repository.
```

The history of a repo

To create a repository in GitHub:

1. login
2. click the plus sign at the top of the page
3. click “New repository”
4. choose the name (usually lowercase, with - for spaces)
5. choose whether you want to keep it private or public
6. click “Create Repository”

The same address you see on the page of a GitHub repo is the same you can use to clone to your computer.

Digression: Cloning a popular repo; open source

Let's see how to clone any publicly available git repo.

`reghdfe` is a popular command created by Sergio Correia to do "regressions with any number of fixed effects" in Stata

He has made the code available on GitHub at https://github.com/sergiocorreia/reghdfe

He has open sourced the code with an MIT license: you can use commercially, modify and distribute it freely.

To clone the entire source code and its history, just run:

```
$ git clone https://github.com/sergiocorreia/reghdfe
Cloning into 'reghdfe'...
remote: Enumerating objects: 44, done.
remote: Counting objects: 100% (44/44), done.
remote: Compressing objects: 100% (27/27), done.
remote: Total 4911 (delta 24), reused 33 (delta 17), pack-reused 4867
Receiving objects: 100% (4911/4911), 12.81 MiB | 1.67 MiB/s, done.
Resolving deltas: 100% (3733/3733), done.
```

The history of a repo

Let's get back to our own empty repo.

Open the folder with the repo name and, before we do anything else, run `git status`

```
$ cd git-tutorial

$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git
   add" to track)
```

This already tells you some information: what branch we are in, that no commits have been made, and there is nothing yet to commit. We'll learn what each of these mean.

The history of a repo

Let's create a `readme.txt` file with some notes. Then, run `git status` again:

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will
     be committed)

        readme.txt

nothing added to commit but untracked files present
   (use "git add" to track)
```

The states of files

Now we'll start learning about the 4 states of files. A file in a repo can be:

1. **Untracked**: the git repo does not include it in commits, save any changes, etc. Default state for new files

This is the status of our `readme.txt` file. It is untracked by the repo.

We want git to track it, so we'll add it with `git add readme.txt`.

You can use `git add -A` to add all changes in the directory

The history of a repo

Let's add the `readme.txt` file to track it and run `git status`

```
$ git add readme.txt

$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   readme.txt
```

Now, git is tracking the file and is noticing that there is something new. We now have a new status for `readme.txt`: staged

The states of files

A file in a repo can be:

1. **Untracked**: the git repo does not include it in commits, save any changes, etc. Default state for new files
2.
3.
4. **Staged**: the changes made at the time you `added` the file (but not newer changes since then) will be included in the next commit

The history of a repo

Let's commit the file to the repo

```
$ git commit -m "Adds readme.txt file"
[master (root-commit) f19930d] Adds readme.txt file
 1 file changed, 1 insertion(+)
 create mode 100644 readme.txt
```

`git commit` adds the `readme.txt` file to the repo in its current version.

`-m` adds the comment in quotes to describe the commit. git will always ask you to provide a comment and you should provide a short, but clear one, usually starting with a verb.

Don't forget to commit as you work. Whenever you find a natural break for the changes you are making is probably a good time to commit the changes you have made and leave a message explaining them. This makes it easier to restore previous versions if you find a bug, or to find the specific line changes that broke your code.

The history of a repo

If we run `git status` now, we see:

```
$ git status
On branch master

nothing to commit, working tree clean
```

There is nothing to commit. There are no new untracked files, there are no tracked files that have been modified since the version stored in the last commit.

Our `readme.txt` file is thus in a new stage: unmodified.

The states of files

A file in a repo can be:

1. **Untracked**: the git repo does not include it in commits, save any changes, etc. Default state for new files
2. **Unmodified**: tracked files that have not been modified since the version stored in the last commit
3.
4. **Staged**: the changes made at the time you `added` the file (but not newer changes since then) will be included in the next commit

The history of a repo

Let's now write something new to our file and run git status again:

```
$ git status
On branch master

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:   readme.txt

no changes added to commit (use "git add" and/or "git commit
    -a")
```

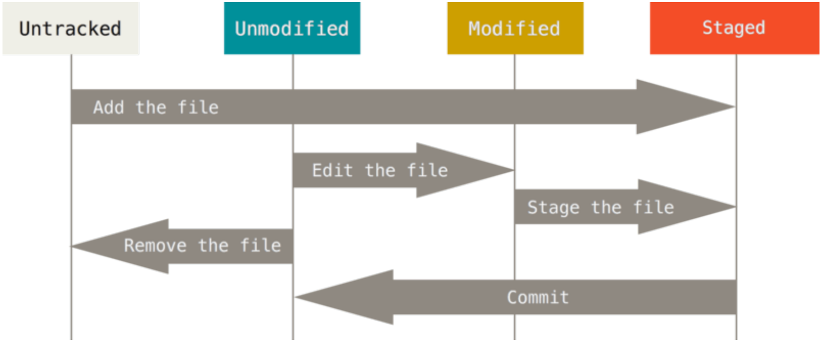
We now see the remaining status of a file: modified. Notice after changing a file, it does not get automatically staged for the next commit. You need to explicitly tell git you want to add/stage those changes with `git add` as we did before.

The states of files

A file in a repo can be:

1. **Untracked**: the git repo does not include it in commits, save any changes, etc. Default state for new files
2. **Unmodified**: tracked files that have not been modified since the version stored in the last commit
3. **Modified**: tracked files that have been modified since the version stored in the last commit
4. **Staged**: the changes made at the time you `added` the file (but not newer changes since then) will be included in the next commit

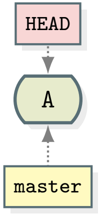
The states of files



Source: https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository

The history of a repo

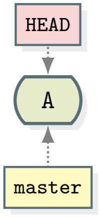
We've created our first commit. This is our repo's graph:



Your commit will have a long hash that works as its identifier. In my case, it was `f19930d4adc474ffa844087f3a89dbeedb260251`, but git is fine if you call it `f19930d`. In the graph, I'll just call it A for simplicity. The hash has two uses:

1. **Identification**: refers to specific commit in project's history
2. **Security**: it's "impossible" to get a repeated hash. It's computed with info on parents, date and time, author and content of changes. Any intentional tampering or accidental corruption would produce different hashes and would be noticed by git

The history of a repo



`HEAD`, as indicated in the glossary, tells git what is the commit that the local machine is working on (checking out) now.

`master` is the default name of the main branch. We will look at branches soon.

The history of a repo

Let's commit the changes in our modified file:

```
$ git add readme.txt

$ git commit -m "Adds a few things to our readme
   file"
[master d280048] Adds a few things to our readme file
 1 file changed, 1 insertion(+), 1 deletion(-)
```

The history of a repo

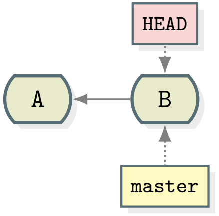
We can use `git log` to see our current history:

```
$ git log
commit d280048e352f5c02d7aa9df521ac1b05785f14bb (HEAD -> 
    master)
Author: Luís Fonseca <author@email.com>
Date:   Tue Apr 24 11:46:50 2018 +0100

    Adds a few things to our readme file

commit f19930d4adc474ffa844087f3a89dbeedb260251
Author: Luís Fonseca <author@email.com>
Date:   Tue Apr 24 11:45:17 2018 +0100

    Adds readme.txt file
```

See https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History for many other options to visualize a repo's history

The history of a repo

Our graph now looks like this:



So, a git repo is a directed (each commit points to its parents) acyclic (there is no way to follow the arrows and come back to the starting point) graph.

`HEAD` now points to the new commit where we are working; so does the branch

Going remote

All the operations we did so far were on our computer. They weren't yet sent to the remote server (GitHub in this case).

This allows you to work freely on a feature before sharing with co-authors. If anything goes wrong, you can always just delete your local repo or branch and the version in the server stays as it is. However, if you want to back up or share with your colleagues the changes you've made, you need to upload to the remote server.

When we created the repo in GitHub and then cloned it empty to the local machine, the remote server was automatically configured. If you first created the repo in your machine with `git init` instead of cloning, you can configure it manually by running:

```
$ git remote add origin
    https://github.com/USERNAME/git-tutorial.git
```

`origin` is the default name of a remote server in git. You can name it differently, and/or have more than one server you upload to

Going remote

So, let's push (upload) our changes to the remote server:

```
$ git push
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 519 bytes | 519.00
   KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/USERNAME/git-tutorial.git
 * [new branch]      master -> master

$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

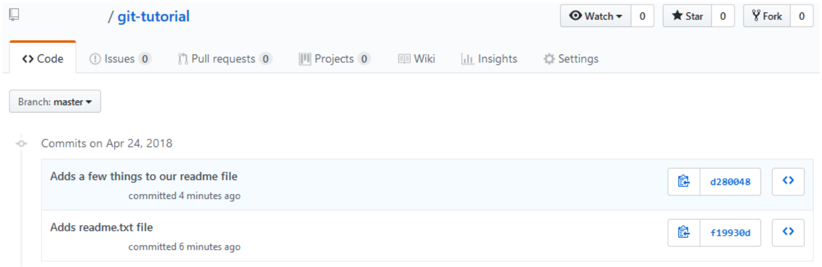
Going remote

So, `git push` uploads the changes to the remote server. Since the remote server was empty, it created the master branch in the remote server and linked your local `master` branch with one with the same name in the server.

Now notice `git status` tells you your local branch is up to date with the `master` branch in the `origin` (the GitHub server)

Important: git is a **decentralized** version control system. Each git repo of the same project is just as valid as any other. Your local repo contains all the history just as the GitHub remote does. If GitHub servers were to explode tomorrow, your local repo would contain all the history of the project up to its last fetch or pull.

Going remote


/ git-tutorial

Branch: master

Commits on Apr 24, 2018

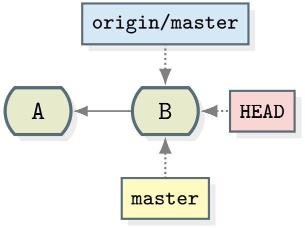
Adds a few things to our readme file
committed 4 minutes ago
d280048

Adds readme.txt file
committed 6 minutes ago
f19930d

Now, the commits show up on the GitHub page

How does our graph look like now?

The history of a repo



We now need a label to indicate where the GitHub / remote / origin master branch is. `origin` is the default name for the remote repo from where you cloned.

The yellow labels will indicate a branch on your local repo (i.e. in your computer)

The blue labels will indicate a branch on the remote repo (e.g. GitHub)

Going remote

Let's now add and commit a new file treat_data.do to our local repo:

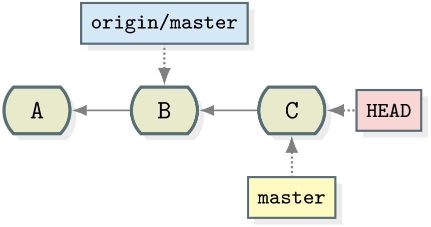
```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will
     be committed)

        treat_data.do

nothing added to commit but untracked files present
   (use "git add" to track)

$ git add treat_data.do
```

Going remote

```
$ git commit -m "Adds a file to treat data"
[master dddf614] Adds a file to treat data
 1 file changed, 1 insertion(+)
 create mode 100644 treat_data.do

$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

The history of a repo

We have commited the file to our local repo but have not yet pushed to the origin



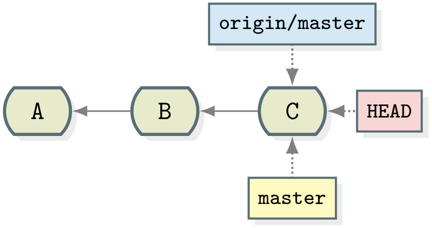
Your `master` branch is now one commit ahead of the master branch in the origin. Until you push it, no one will see what you changed.

Going remote

Now, if we push the new commit:

```
$ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 319 bytes | 319.00
   KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/USERNAME/git-tutorial.git
   d280048..dddf614  master -> master

$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

The history of a repo



Now, the origin (remote repo) is up-to-date with our local repo.

Remote changes

We've seen how git works when you do changes locally and want to push them to a server.

What if changes were made:

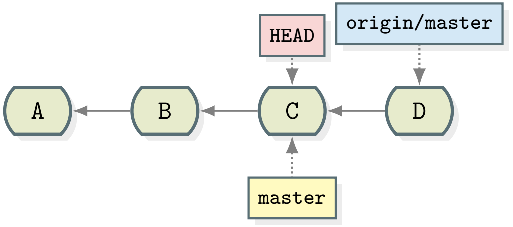
1. by you in a different computer
2. by you on GitHub
3. by a co-author

Edit the `readme.txt` file in GitHub to simulate a co-author making changes. Open the file there, click on the pencil, write additional text, and commit to the master branch.

How does our graph look like?

The history of a repo

We have committed the file to origin but not to our local repo:



The history of a repo

What happens if we run git status?

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

Why does it say that the branch is up to date?

git is decentralized (designed to work locally). It is not in permanent communication with the remote repository.

The history of a repo

How do we tell git to go and see if the remote repo has changed?

```
$ git fetch
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0),
   pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/USERNAME/git-tutorial
   dddf614..aba9c99  master     -> origin/master

$ git status
On branch master
Your branch is behind 'origin/master' by 1 commit,
   and can be fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

The history of a repo

`git fetch` asks the remote for all the commits it has and downloads this information. However, as you can see with `git status`, it does not immediately update your local repo with this.

`git status` is telling us that the local branch is 1 commit behind and can merged with the remote with fast-forward. Fast-forward means that git has found the common ancestor between local and remote and all changes build on top of that commit, so no complicated merge is necessary: just apply the changes.

We can then use `git merge` to update our local repo. But in general, you'll do the entire procedure at once with `git pull`

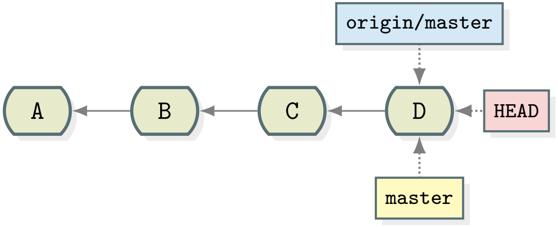
The history of a repo

git pull combines the fetch and the merge and immediately updates your local repo with the one in the server.

```
$ git pull
Updating dddf614..aba9c99
Fast-forward
 readme.txt | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)

$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

The history of a repo

Both are up to date with each other now:



Branches

- Branches are a great feature
- They allow you to work in different things in the project simultaneously, and only integrate them when they're ready
- You can keep a stable version of your results/simulations in one branch, while experimenting with new things in another branch independently
- Allows you to not have to rename dependencies, create new copies of files, etc., when experimenting
- Can easily change back and forth between different versions of the project you are working on, keeping your folders clean and without endless and costly (in disk/cloud space) backups

Branches - an example

- You are working on a project. A referee asks you to do a complicated new simulation that will take weeks
- You're halfway done, but then you are reminded of a conference deadline that requires some minor changes to the previous stable version of the draft and results
- Without version control, you will easily have old and new results, and old and new codes mixed up
- If when starting the simulations you create a new branch called `new_simuls`, you can work there without any issue.
- As the conference approaches, you return to your `master` branch with the stable version of the paper; do whatever you need there
- Then, return to the `new_simuls` branch, continue working there and integrate in the master branch when it's ready.
- git easily shows you what changed between different branches, makes it easier to solve conflicts when they arise, etc.

Branches

Let's create a new branch called `newregs` :

```
$ git branch newregs

$ git checkout newregs
Switched to branch 'newregs'

$ git status
On branch newregs
nothing to commit, working tree clean
```

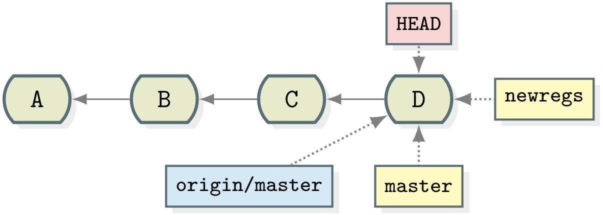
`git branch <name_of_branch>` creates a new branch, but does not change your working directory to that branch.

To change your working copy there, you need the command `git checkout <name_of_branch>`. These two commands can be done in a single line by doing `git checkout -b <name_of_new_branch>`

Note that `git status` now tells you you are in the `newregs` branch

Branches

How the tree looks like now:



The branch only exists in your computer for now. No co-author will see it yet: thus, we only have the yellow label for the branch.

Also, since you haven't made any new commit to the new branch, it points to the same commit as `master`.

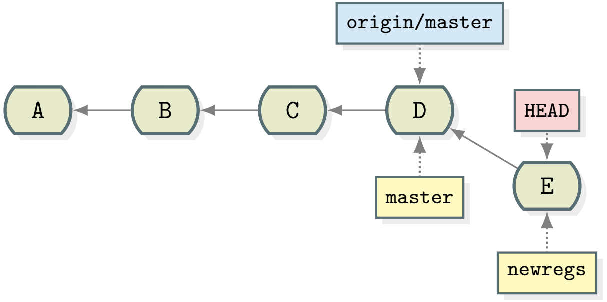
Branches

Let's add and commit a file `new_regs_data.do` in the new branch

```
$ git status
On branch newregs
Untracked files:
  (use "git add <file>..." to include in what will
     be committed)

        new_regs_data.do

nothing added to commit but untracked files present
    (use "git add" to track)

$ git add -A
$ git commit -m "Adds new regs data do file"
[newregs 17bd274] Adds new regs data do file
 1 file changed, 1 insertion(+)
 create mode 100644 new_regs_data.do
```

Branches

How the tree looks like now:

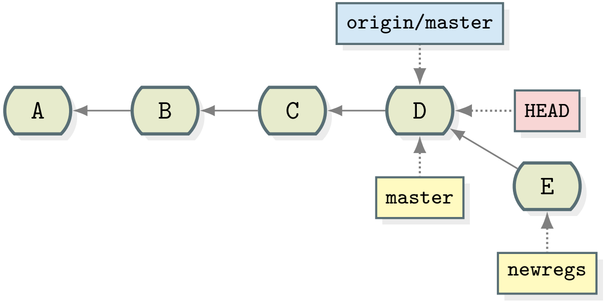


Branches

We can easily go back and forth between branches. As we do so, the folder in your computer changes accordingly:

```
$ git checkout master
Switched to branch 'master'
```

Notice that now HEAD points to commit D

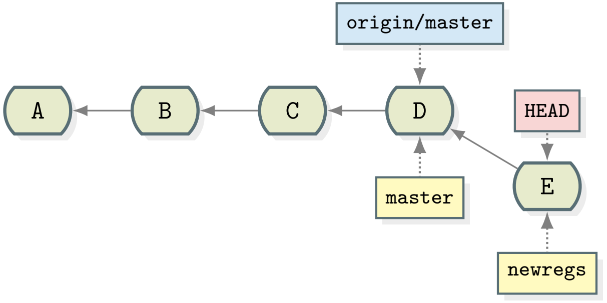


Branches

We can easily go back and forth between branches. As we do so, the folder in your computer changes accordingly:

```
$ git checkout newregs
Switched to branch 'newregs'
```

Notice that now `HEAD` points to commit `E`



Branches

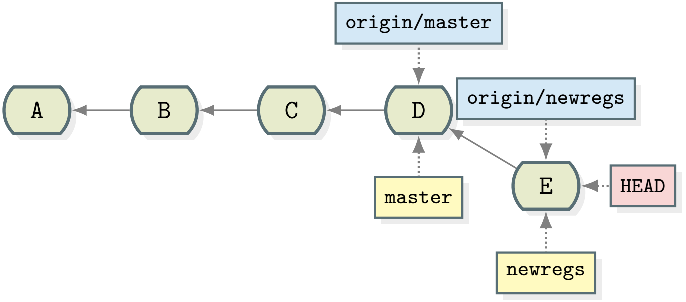
Let's now push the new branch to the remote server.

```
$ git push --set-upstream origin newregs
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 349 bytes | 349.00
   KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/USERNAME/git-tutorial.git
 * [new branch]      newregs -> newregs
Branch 'newregs' set up to track remote branch
   'newregs' from 'origin'.
```

In the first time, we need to add info to `git push` since the origin had no `newregs` branch yet. The command tells git to send the current branch to the origin (GitHub) to a branch also called `newregs`. Next time we push to this branch, `git push` is enough

Branches

How the tree looks like now:



Branches

Suppose we now need to work again on the master branch before we have finished the work in new regs. We will change the `readme.txt` file again

```
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

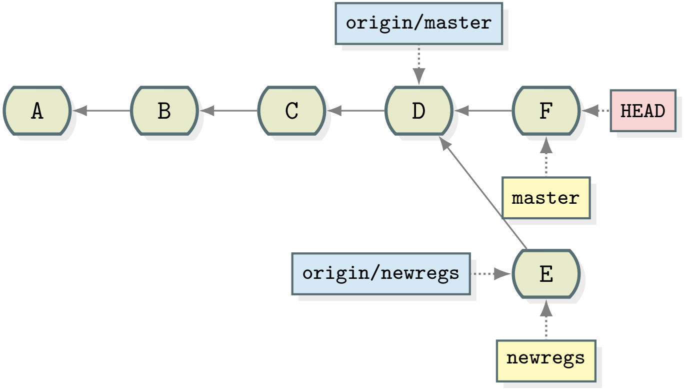
Notice the `new_regs_data.do` file disappears from the folder.

At this point, make some change to the `readme.txt` file and:

```
$ git add readme.txt

$ git commit -m "Updates readme file"
[master 7beccc9] Updates readme file
 1 file changed, 2 insertions(+)
```

Branches

How the tree looks like now:



Branches

Suppose you are happy with your work in `newregs` and want to integrate it in the master branch. We can use the merge command to merge the two branches.

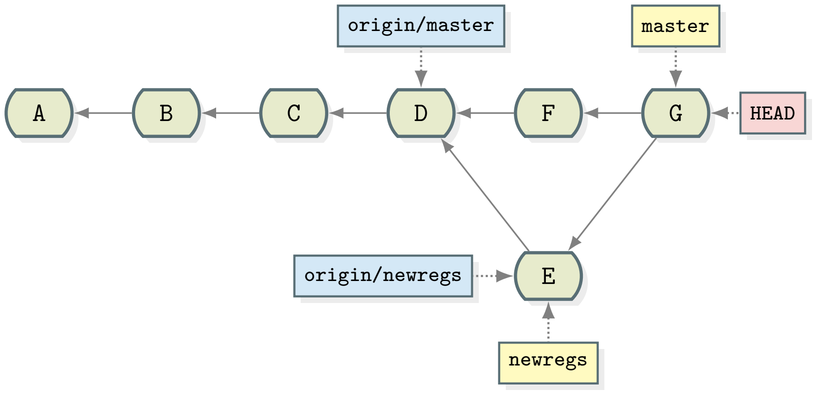
Since the changed affect different files, no conflicts will occur. There are cases where there are conflicting changes. git has tools to help you solve this, but I won't cover those cases here.

Ensure you are checking out the branch you want to merge **into**:

```
$ git checkout master
Already on 'master'
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

$ git merge newregs -m "Integrates newregs"
Merge made by the 'recursive' strategy.
 new_regs_data.do | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 new_regs_data.do
```

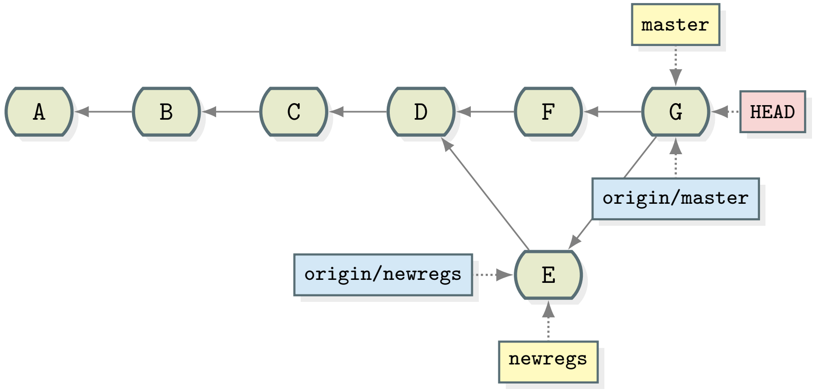
Branches

How the tree looks like now:



Branches

We can now use `git push` to push this to the remote:



You can now just leave the old branch there, or delete it. When you delete a branch in your local repo, it does not get deleted in the remote repo. You need to do that specifically.

Sourcetree GUI

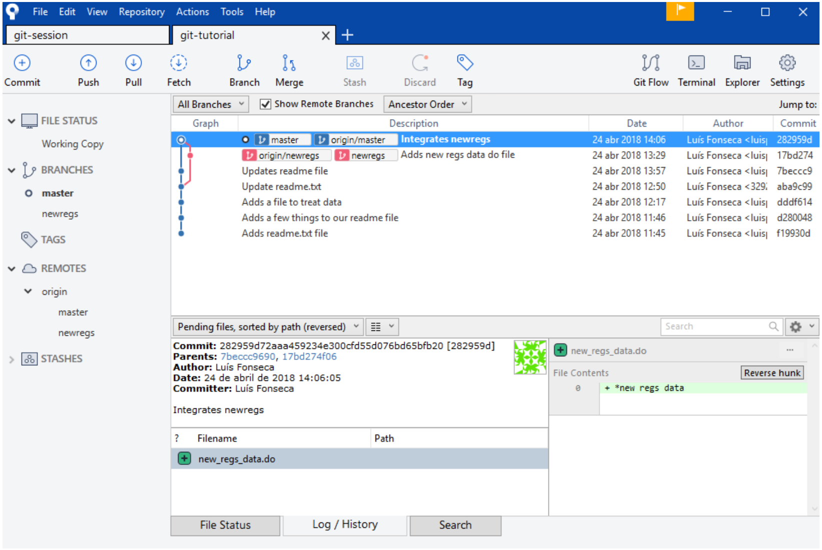
After you get comfortable with commands, GUIs like Sourcetree make many things things easier:

- Change between different versions (commits) of the project
- See differences between versions, changes since last commit
- Specify large groups of files but not all to be committed
- Specify lines within files to be committed separately
- Revert changes
- Delete local and remote branches
- See the repo graph

But they only do what git commands allow, so it's useful to understand what are they doing in the background by being comfortable with the command line as well

Sourcetree GUI

This is how Sourcetree shows the current state of the repo



Graph	Description	Date	Author	Commit
	master origin/master Integrates newregs	24 abr 2018 14:06	Luís Fonseca <luis|	282959d
	origin/newregs newregs Adds new regs data do file	24 abr 2018 13:29	Luís Fonseca <luis|	17bd274
	Updates readme file	24 abr 2018 13:57	Luís Fonseca <luis|	7beccc9
	Update readme.txt	24 abr 2018 12:50	Luís Fonseca <329:	aba9c99
	Adds a file to treat data	24 abr 2018 12:17	Luís Fonseca <luis|	dddf614
	Adds a few things to our readme file	24 abr 2018 11:46	Luís Fonseca <luis|	d280048
	Adds readme.txt file	24 abr 2018 11:45	Luís Fonseca <luis|	f19930d

Commit: 282959d72aaa459234e300cfd55d076bd65bfb20 [282959d]
Parents: 7beccc9690, 17bd274f06
Author: Luís Fonseca
Date: 24 de abril de 2018 14:06:05
Committer: Luís Fonseca

Integrates newregs

?	Filename	Path
+	new_regs_data.do	

new_regs_data.do

```
0    + *new regs data
```

Undoing things

- Forgot to include a file in the last commit

```
git commit -m 'initial commit'
git add forgotten_file
git commit --amend
```

  with `-m "New description"` or `-no-edit` for same description

- Unstage a staged file

```
git reset HEAD staged_file.txt
```

- Clear the staging area / index

```
git reset HEAD -- .
```

- Unmodifying a modified file

```
git checkout -- staged_file.txt
```

Undoing things

- Revert changes in commit, leaving a message

  ```
  git revert <commit_hash>
  ```

  creates a new commit reverting the changes of the hash you specify. Only applies to that commit, not the following ones. Use `git revert <oldest_commit_hash>..<latest_commit_hash>` for range.
- Revert pushed changes without leaving trace (Careful!)

  ```
  git reset <previous label or hash>
  git push -f
  ```

  `git reset` returns repo to specific commit. Then, `-f` in push forces this to be pushed to the server, which overwrites all the children commits. This destroys data, so should be used only if you know what you're doing. Also, use only on repos that you work alone and if the commits have not been pulled by any other machine or pushed to remote, otherwise things can get messy.

Undoing things

- General tips for resetting without leaving trace in history
  - Easier and safer if you haven't yet pushed to the remote repo
  - If pushed but you are working alone, easier (but still be careful)
  - If pushed in a repo with co-authors who may have pulled the changes to their computers, very risky. Easier to just revert changes and leave the message than to rewrite history
- Reset branch to a commit
  - https://git-scm.com/book/en/v2/Git-Tools-Reset-Demystified provides a good description of the powerful `git reset` command
  - Google for specific cases, and learn as you go along
  - GUIs can make this easier
  - Learn the difference between soft, mixed and hard reset

Additional useful things to know / explore

- git tracks renames if content remains mostly the same initially
- A `.gitignore` file in the repo tells git to ignore folders, file types and names. Patterns are very flexible. See https://www.gitignore.io/ for examples
- Git Large File Storage (LFS) replaces large files with text pointers in the repo, storing the files on another server
  - Avoids having to store large files in the repo, which can slow it down when cloning, committing
- `git tag` tags the current commit (e.g. `git tag -a v1.4 -m "my version 1.4"`). Easier to find older versions without remembering or writing the hash. Tags are local unless pushed to remote with `git push origin -tags`.
  - GitHub has a nice page to create tagged versions and stores all files in a zip for each tagged version.
- `git diff` allows you to see the difference between commits
- `git stash`: when you need to switch branches while in the middle of other work that you do not want to commit now but want to return to later. See documentation page

Useful references

- General good practices by Gentzkow and Shapiro
  - Code and Data for the Social Sciences: A Practitioner's Guide
  - (they weren't using git yet, but do discuss version control)
  - https://github.com/gslab-econ for their GitHub repos
    - https://github.com/gslab-econ/ra-manual/wiki explains their practices for their RAs
- git stuff
  - Pro Git book (https://git-scm.com/book/en/v2) is the best and clearest reference book
  - Git documentation available at https://git-scm.com/docs
  - https://try.github.io/is a very introductory tutorial
  - http://swcarpentry.github.io/git-novice/ is an excellent hands-on tutorial
- Other sources I borrowed from:
  - https://betterexplained.com/articles/a-visual-guide-to-version-control
  - https://michaelstepner.com/blog/git-vs-dropbox
  - "git for ages 4 and up" by Michael Schwern (https://www.youtube.com/watch?v=1ffBJ4sVUb4)