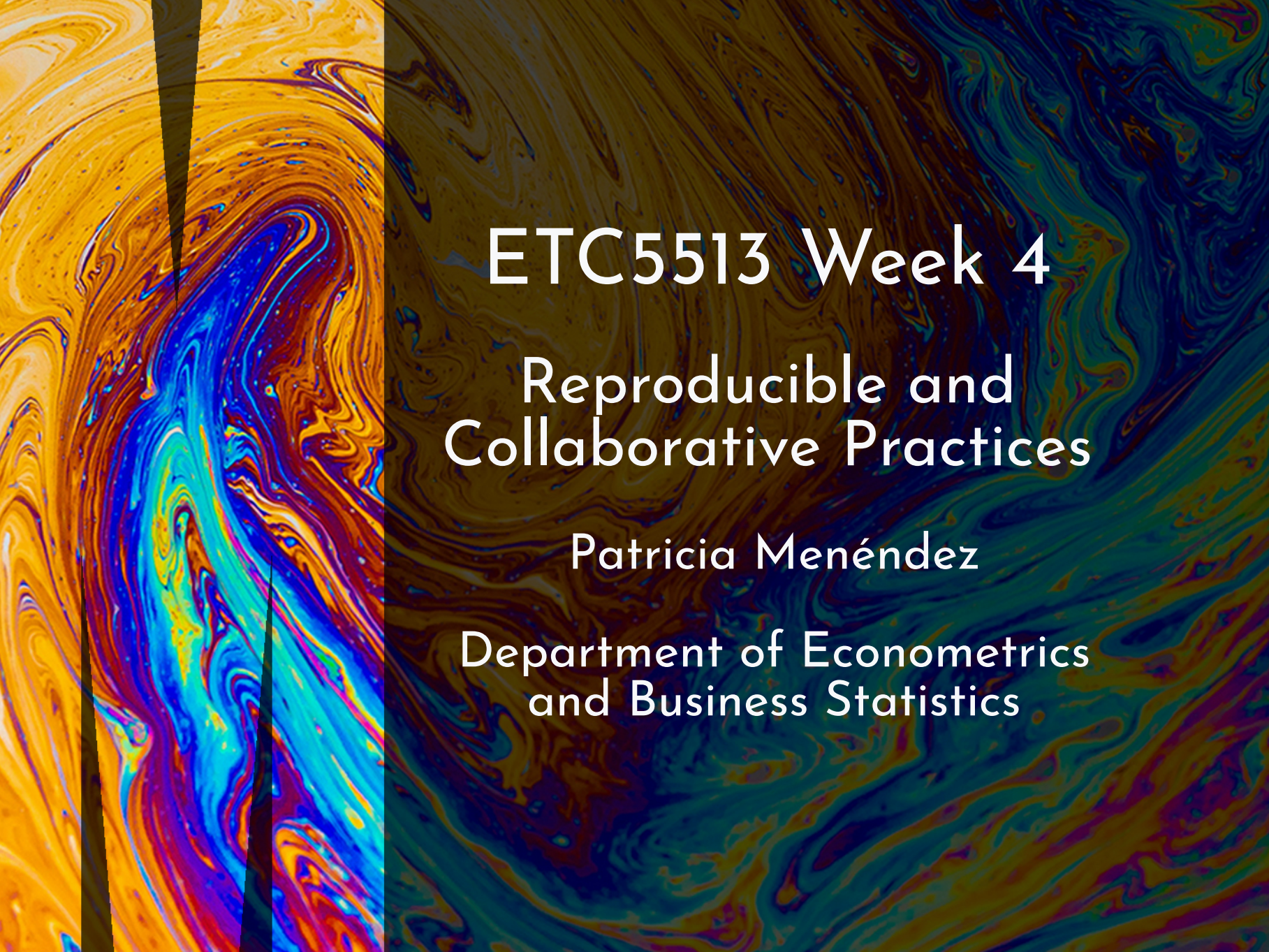


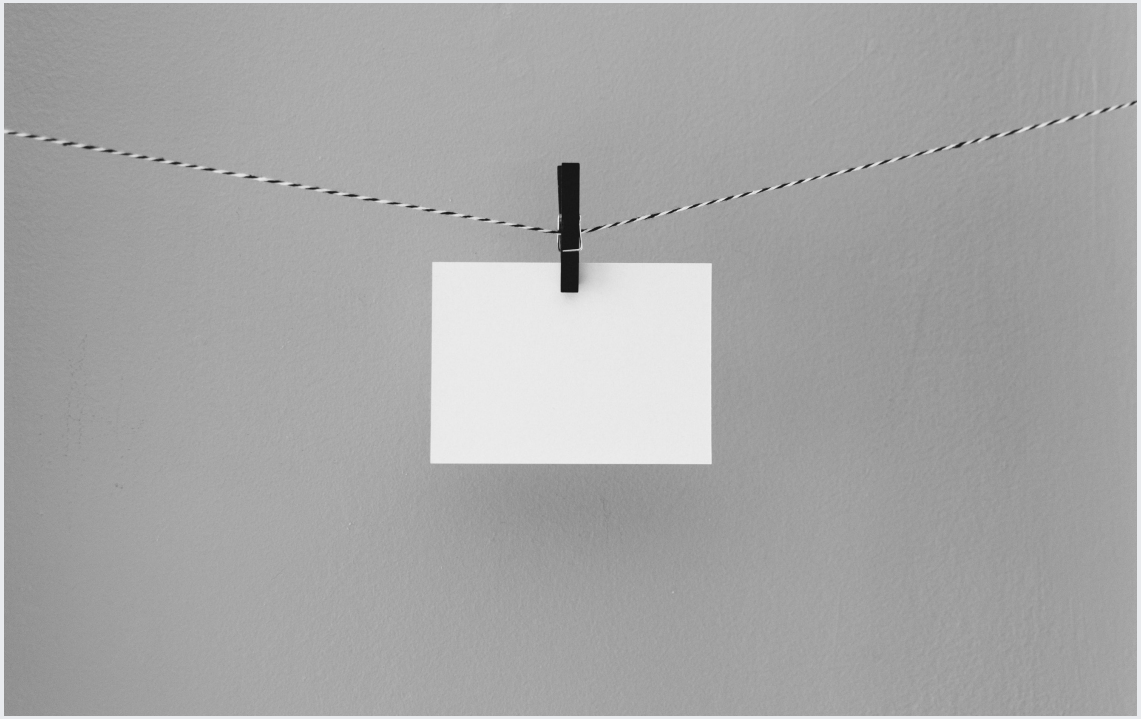
ETC5513 Week 4

Reproducible and Collaborative Practices

Patricia Menéndez

Department of Econometrics and Business Statistics

Open frame



Recap

Recap: In week 3

1. Motivation for version control
2. Git
3. Introduction to command line
5. Github
6. Integration between Github and Rstudio
7. Workflow for using version control

This week (Week 4)

1. **Learn more on creating reproducible reports:**
    - Referencing
    - Learn about package Bookdown
    - Talk about css files
2. **More on Git:**
    - Create and delete branches
    - Merge branches
3. **Solving Git conflicts**
4. **Please make sure you install VS code as an editor for writing longer commits**
5. **GitKraken as a GUI to work with Git/GitHub**

Connecting the dots

So far:

- Learned to create basic reproducible reports using R
- Learned how to connect our reproducible reports to Git and GitHub (version control)

Next:

- Need to learn how to make "professional reports" not only on html but also in pdf
- How to collaborate on projects with other colleagues
- Learn how to solve issues on GitHub

Learning more about creating Rmd documents

When we have figures or plots in our reports it is a great idea to set up some global options at the beginning of our document:

Keeping our R figures inside a folder

```
---
title: "My Report"
author: "Patricia Menéndez"
output:
  html_document:
  keep_md: true --> Allow us to keep figures
---
```

Displaying figures

Options inside the R code chunks:

- fig.align --> alignment of the figures in the report with options default, center, left, or right fig.alig = "center"
- fig.cap --> captions fig.cap = “My Amazing Graph”
- fig.height & fig.width --> size of the figure in inches fig.height = 7
- out.height & out.width --> size of your plot in the final file. For example out.width = "50%" which means half of the width of the image container (if the image is directly contained by a page instead of a child element of the page, that means half of the page width).

More on these controls here

Inserting figures

- Using markdown syntax:

```
![Caption](path-to-image-here)
```

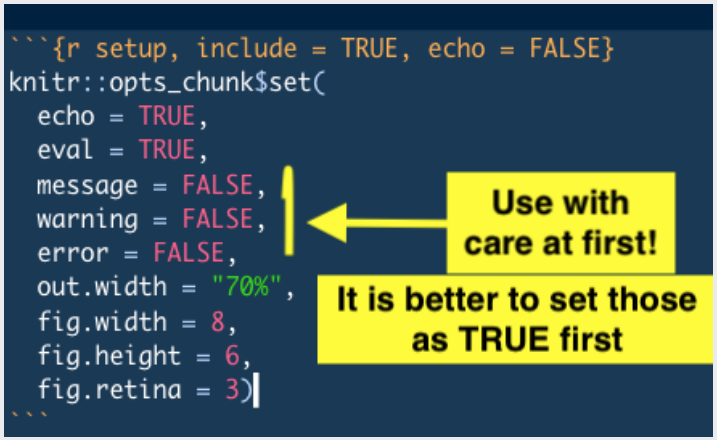
- Using Knitr package syntax (this is my prefer option)

R code chunk options -->

```
```{r out.width = '100%', eval = FALSE}
knitr::include_graphics("figs/rmarkdown.png")
```
```

Setting up global options for our repor.

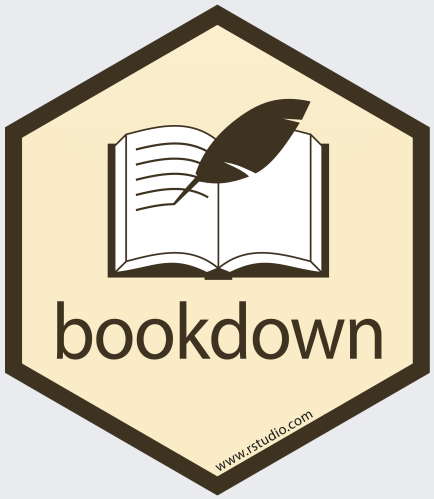
Global options are those that are applied to the entire document:



Best is to add this R code chunk at the beginning of the document before the libraries R code chunk. They can be overwritten by the individual R code chunk options!

Knitr reference guide here

Bookdown R package



- Allow us to reference figures
- Tables
- Articles

The bookdown package is an R package that facilitates writing books and articles/reports with R Markdown.

More info about Bookdown package here

Adjusting YAML: Referencing

In order to use referencing in Rmd documents we need to update the YAML to include the option bookdown::html_document2 as follows:

```
---
title: "My Report"
author: "Patricia Menéndez"
output:
  bookdown::html_document2
---
```

Referencing

Depending on the type of document you want to render, you will need to add the following:

- For html:

  output:
  bookdown::html_document2

- For pdf:

  output:
  bookdown::pdf_document2

- For Word:

  output:
  bookdown::word_document2

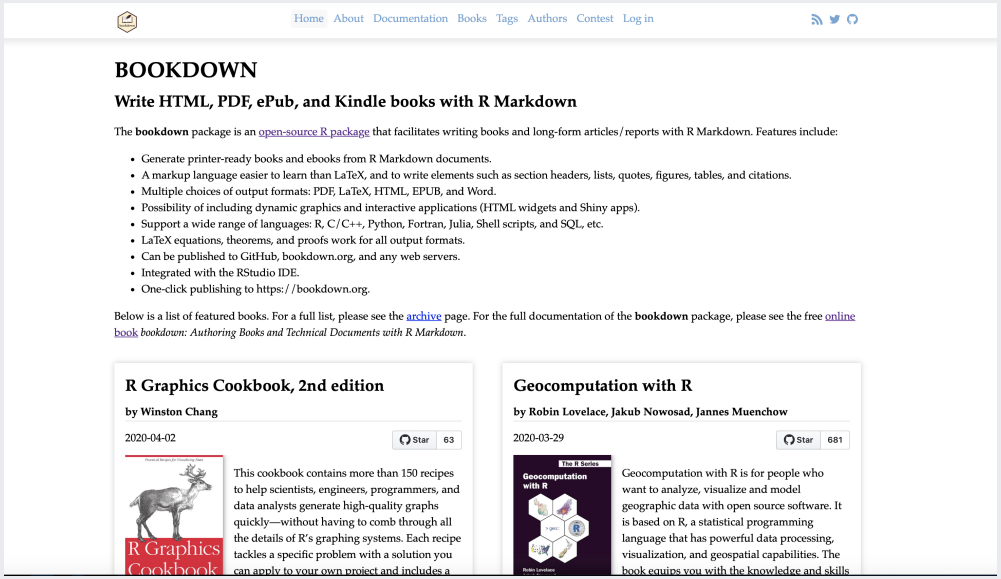
Including referencing and keeping figures

```
---
title: "My Report"
author: "Patricia Menéndez"
output:
  bookdown::html_document2
keep_md: true
---
```

- Inside a folder with filename_files --> figures will be named using the R code chunk names (remember to name your R code chunks!)
- Alternatively, we can add the following option into your Rmd global option set up : fig.path = "Images/" (Refer to slide 10)

This will create a new folder called Images and will place all the figures inside.

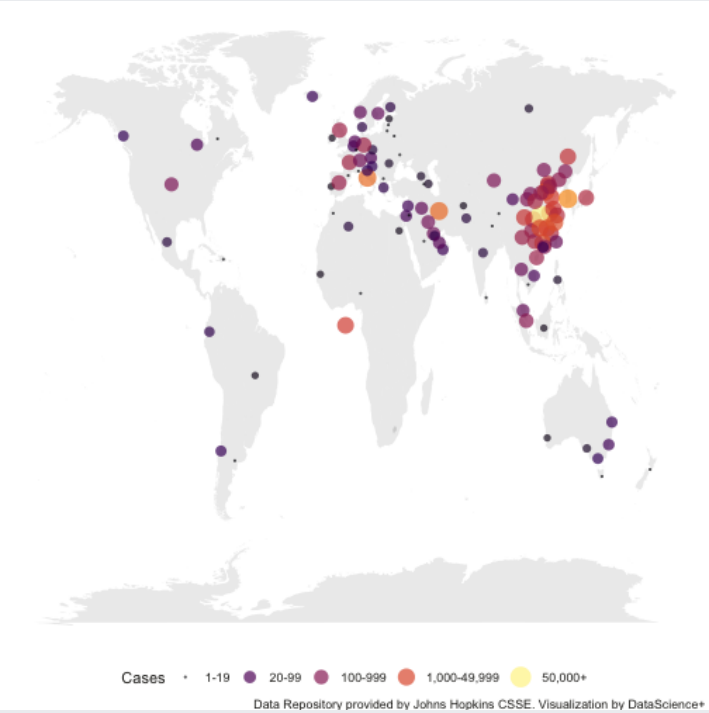
List of books produced using Bookdown



List of books produced with bookdown --> also interesting references

Figure referencing

We need to add the name for the R code chunk and the caption text (You can find the source code for the figure by Anisa Dhana here): {r COVIDmap, fig.caption = "Caption for the figure"}

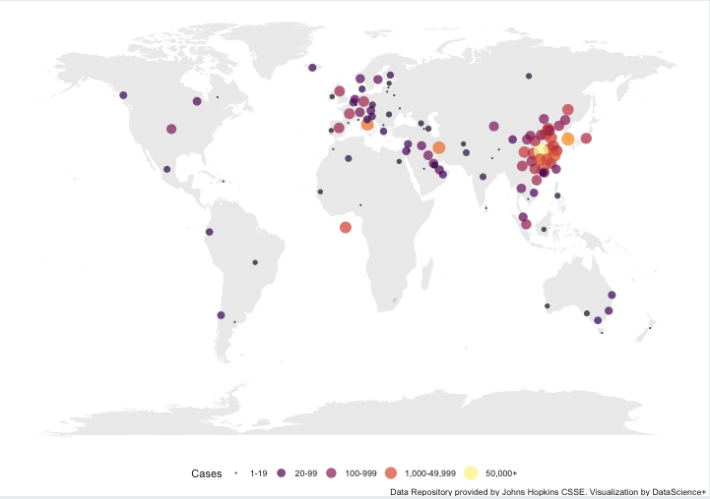


Referencing figures

For the development of the COVID pandemic see Figure 1 ← what we see from the Rmd rendered document (html/pdf etc)

The code above ✅ is produced with the markdown code below 🖐:

For the development of the COVID pandemic see Figure \@**ref(fig**:COVIDmap)) ← what we write in the Rmd file!



Referencing a table

To cite a table we write the following:

Table 1 → Table \@ref(**tab**:chunk-name))

- Remember to create a table we need to organize our data in a data frame or a tibble
- We can use the kable() function from *knitr* package or use the kableExtra package.
- Remember to add the caption inside the kable() function!

Referencing a section

For that we are going to use → \@ref(label)) and we need to include the label in our markdown header for the section:

```
# Section 1 {#label}
```

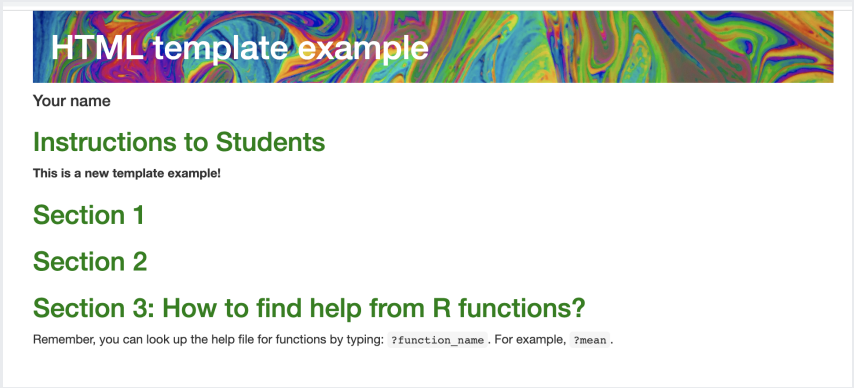
Then in our text we can refer to the section as "In Section \@ref(label)"

Referencing demo

Html reports templates

- Templates can be modified by changing YAML options
- There are lots of available Rmd templates
- YAML can be further modify by using css files. More info here.

Let's have a look at an example that should be familiar for you!



Let's learn more about Git

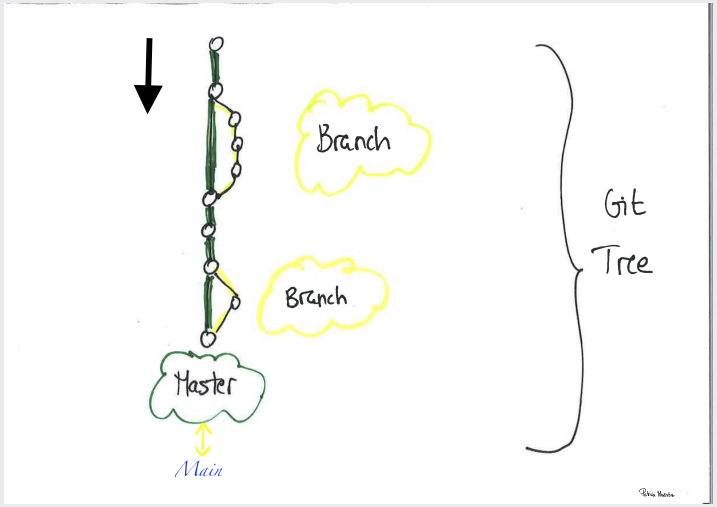
Recap

- git clone is used to target an existing repository and create a clone, or copy of the target repository.
- git pull is used to fetch and download content from a remote repository and immediately update the local repository to match that content.
- git status displays the state of the working directory and the staging area
- git add file_name adds a change in the working directory to the staging area
- git commit -m "Message" (m = message for commit. The git commit is used to create a snapshot of the staged changes along a timeline of a Git projects history.)
- git push origin branch name is used to upload local repository content to a remote repository.

Branching

Each repository has one default branch, and can have multiple other branches. Branching is a great feature of version control!.

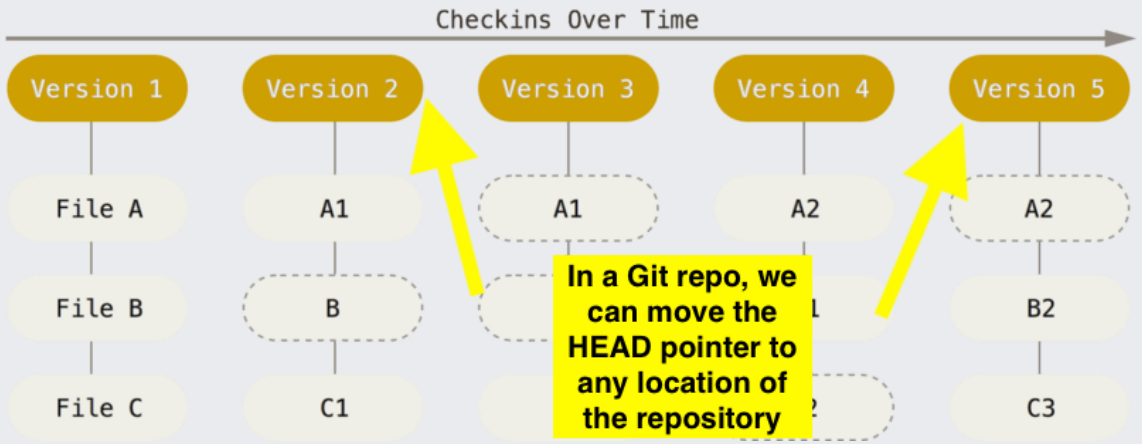
- It allows you to duplicate your existing repository
- Use a branch to isolate development work without affecting other branches in the repository
- Modification in a branch can be merged into your project.



Branching is particularly important with Git as it is the mechanism that is used when you are collaborating with other researchers/data scientists.

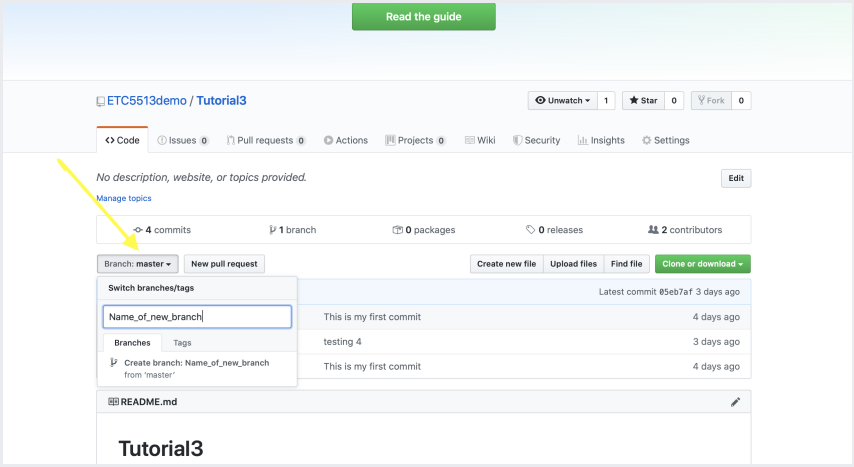
HEAD

HEAD is a pointer that Git uses to reference the current snapshot that we are looking at.



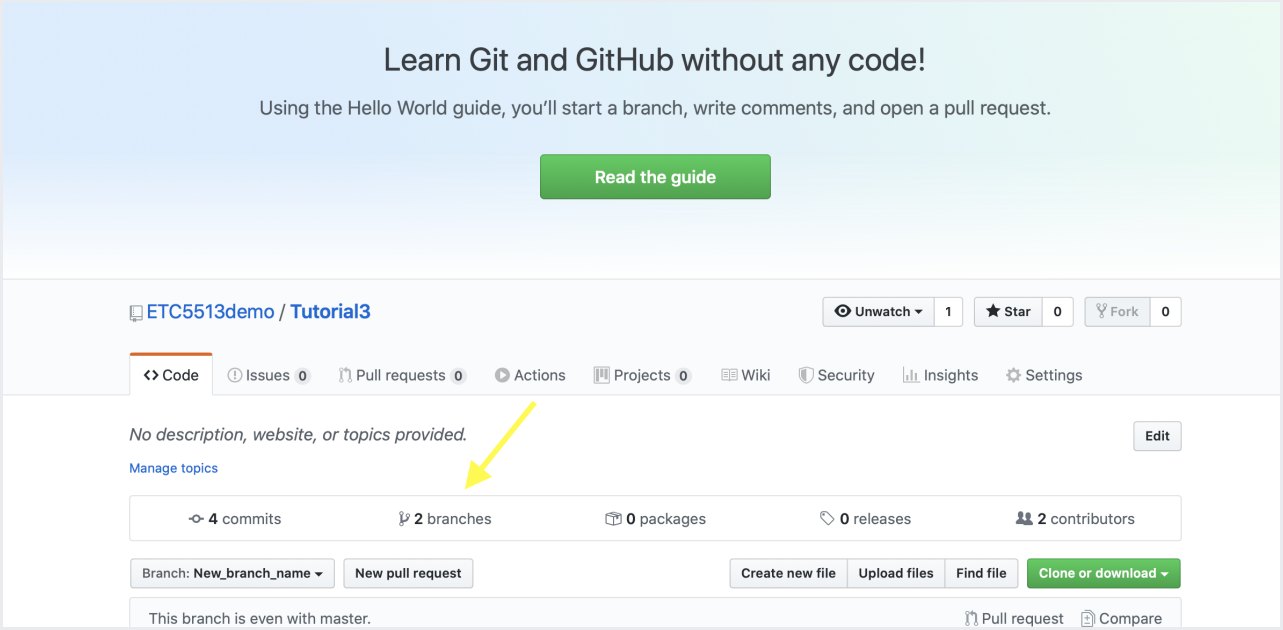
Creating branches from GitHub

- Create branches directly on GitHub. More info here

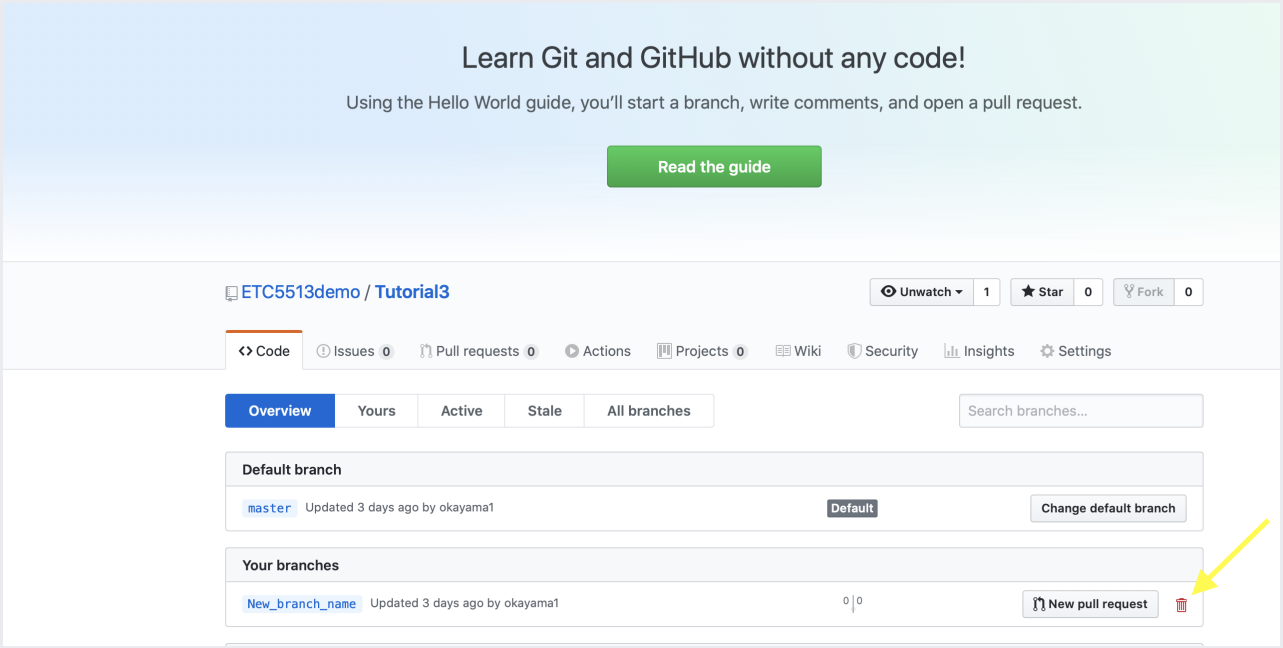


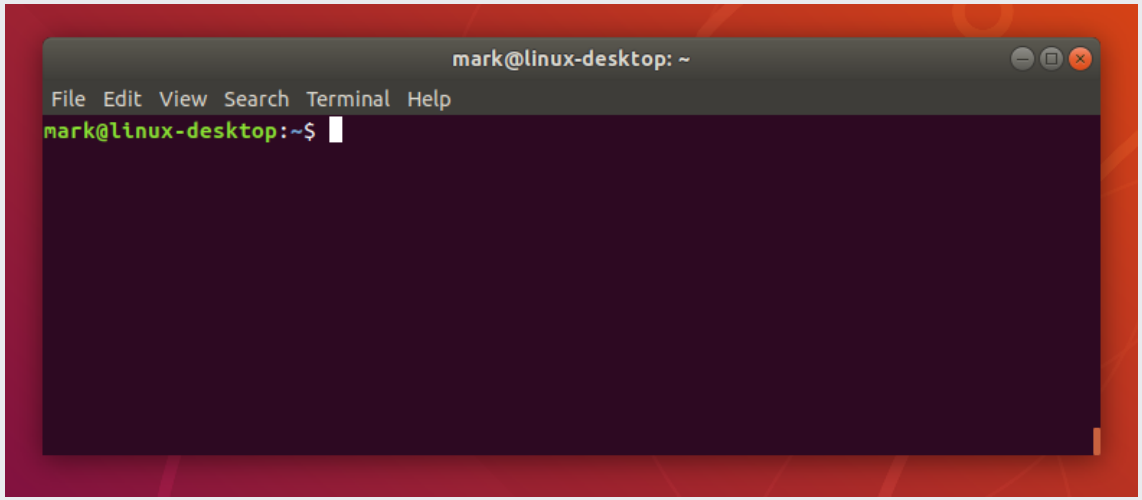
Deleting branches from GitHub

Branches



Deleting branches from GitHub





We will be using our command line interface/Terminal or Git Bash to create and move across branches.

Create branches using the Terminal/Shell/CLI

Using git branch and check out command

- git branch --> show us the branches we have in our repo and marks our current branch with *
- git branch newbranch_name --> creates a new branch but does not move the HEAD of the repo there.
- git checkout newbranch_name --> moves the HEAD to newbranch_name

Git HEAD and checkout

How does Git know what branch you're currently on?

By using the pointer --> HEAD. In Git, this is a pointer to the local branch you are currently on.

Internally, the git checkout command updates the HEAD to point to either the specified branch or commit.

Another way to create and checkout branches

Using check out command

- git checkout -b newbranch_name --> creates a new branch and moves the repo HEAD to this branch
- You can confirm it by using git branch to see in which branch you are currently in

- Checking out a branch --> updates the files in the working directory to match the version stored in that branch
- It tells Git to record all new commits on that branch.

Updating those new branches in the remote repo in GitHub

- We can just update the empty branch into GitHub by git push origin newbranch_name

Alernatively if we had files or changes added into that branch:

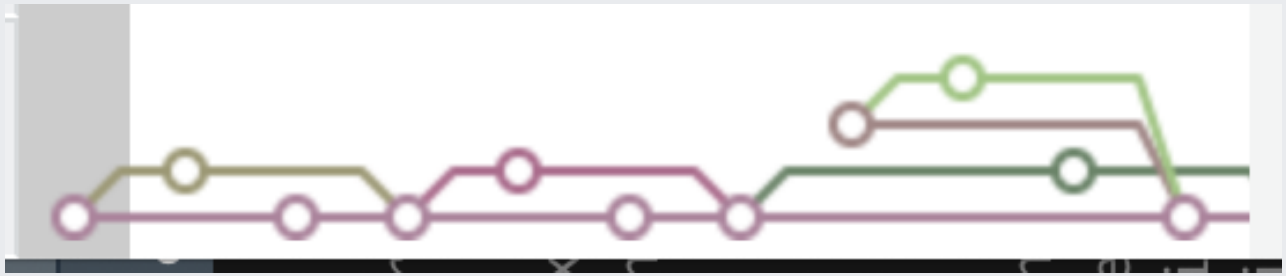
- git add . (--> adding all the modified files into the staging area)
- git commit -m "Updating new newbranch_name"
- git push origin newbranch_name

Merging branches

Switch to the branch that you want to add the stuff into (let's say that is main). Then

- Merge changes into main --> git checkout main
- Anytime we can use git status --> to check the status of our repo
- git merge newbranch_name -m “Merging branches”
- git push origin main updating the remote repository too

Avoiding confusion when creating branches



- Make sure where your branch is starting from
- Branches of branches are, in general, not a good idea

Check out before creating new branch

It is essential to git checkout before creating a new branch.

- If the branch where you are currently working was already merged with the main branch → you'll need to undo almost all the changes from the old branch that did not make it into the main branch.
- Reason: all the old changes from that branch will appear as new changes in combination with the changes that are actually new → It is a mess that you want to avoid!

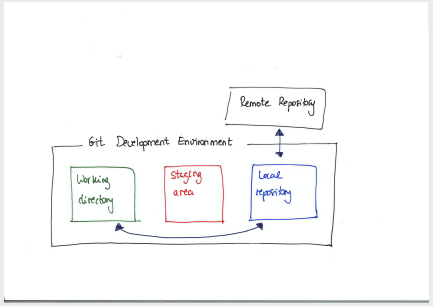
Don't create branches from a branch that is not the main branch unless you are deliberately doing it

Deleting branches using cli

.Deleting branches from your **local** repository

- git branch -a → list all the branches
- Move to main [git checkout main]
- Delete unwanted branch git branch -d Name_of_branch → delete branch called Name_of_branch

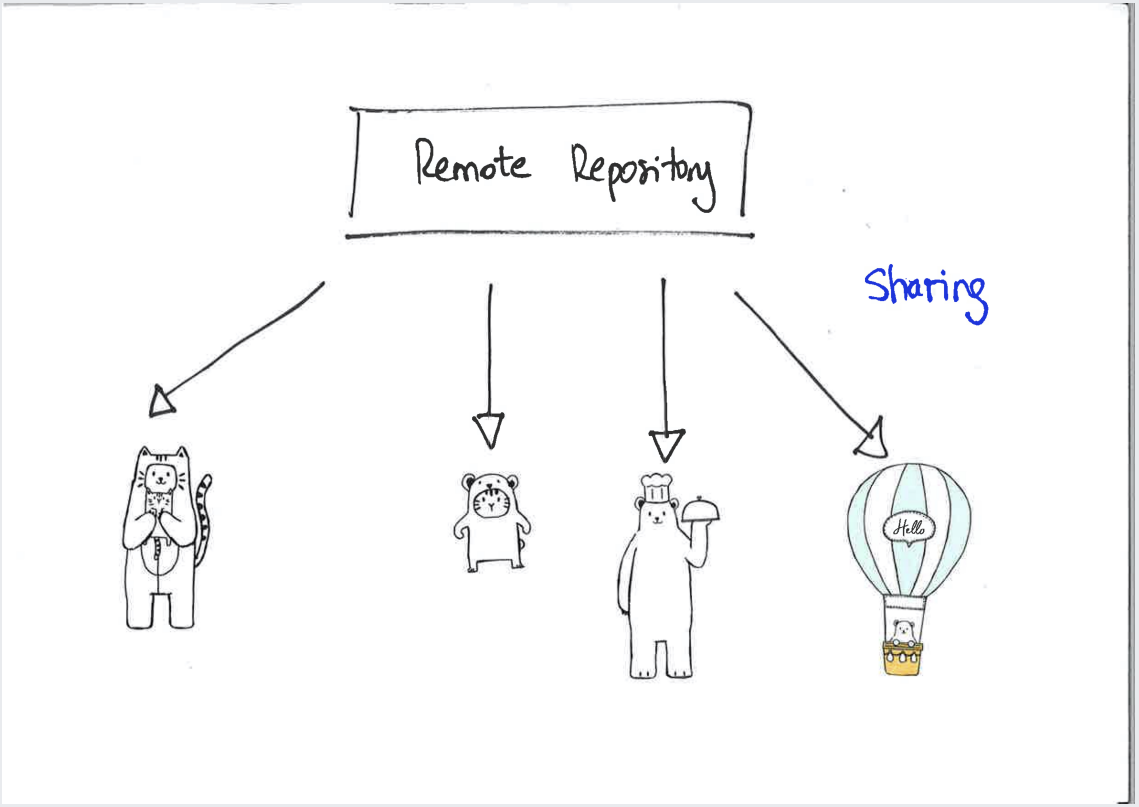
You cannot delete a branch if your HEAD is on that branch



Deleting branches using cli

Deleting branches from your **remote repository** (GitHub)

1. git push origin --delete Name_of_branch



More on Branching

Imagine that you are working on your local repository and a collaborator has created a new branch in your remote repo. You are currently working on your local repo and want to have a look at the new branch. That means that the local repo and your remote repo have diverged. That is, both the local and remote repositories are **not currently synchronized.**

- To synchronize your work git fetch origin.
- git fetch origin looks where “origin” is and fetches any data from it that you don’t yet have.
- It also updates your local database repo, moving your origin/main pointer (HEAD) to its new, more up-to-date position.

About remotes

Note: If the git repo contains more than one remotes, such as there has remotes origin and upstream git fetch will fetch all the changes from the remotes origin and upstream git fetch origin will only fetch the changes from remote origin

Fetch workflow

1. git fetch --> all remote branches
2. Good practice --> Check branches available for checkout
3. Make a local working copy of the branch

Workflow

- git remote --> origin (The git remote command lets you create, view, and delete connections to remote repositories.)
- git fetch origin --> fetch the changes from remote origin (Fetching is what you do when you want to see what everybody else has been working on in the remote repo)
- git branch -a --> all the branches available in the local repository + all the branches fetched from the remote. The branches fetched from the remote origin would be preceded by remotes/origin/

Etiquette for working on someone else branch

- To work on someone's branch --> make a local copy of it
- Work on your local branch (new branch)

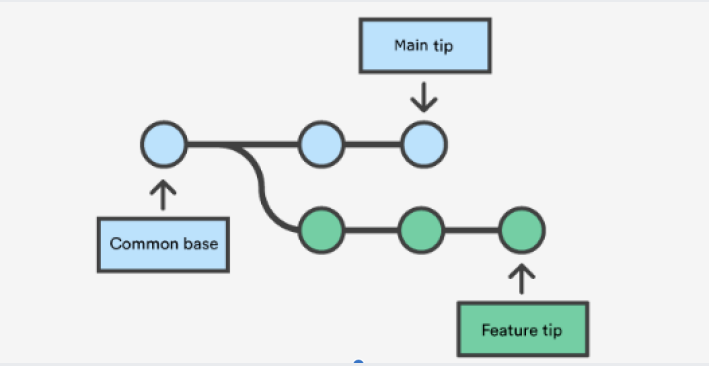
- Then push that new branch to the remote repository

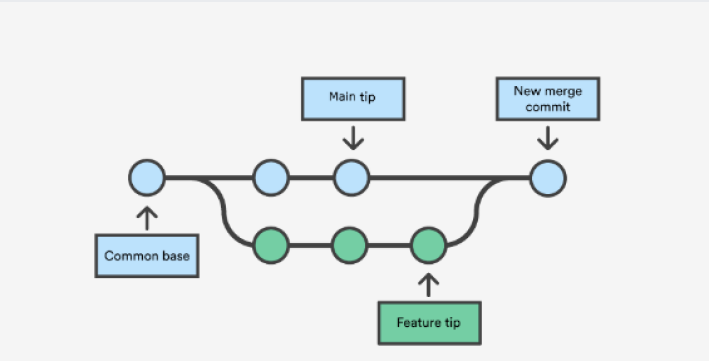
  - To do that:
    - First make sure you are working in that branch in your local repo git branch -a
    - Add changes into the staging area, commit and push changes to the corresponding branch into the remote repository: git add files, git commit -m "Message, git push origin name-of-the-branch

How to go back to your previous branch?

- git checkout branchname
- Imagine that you have two branches:
  - main
  - Alternative_analysis
- To check in which branch you are currently --> git branch or git branch -a --> you will see an * to let you in which branch the HEAD of your repository currently is.
- To go back to main branch (assuming that you were in there) git checkout main

Merging diverging branches





Resource here.

Merging branches sucessfully

Suppose we have two branches: main and new_development and our goal is to bring changes from the branch new_development into our main branch:

1. For merging --> go to main --> branch git checkout main
2. git merge new_development
3. git push origin master
4. This will incorporate the changes made in the branch new_development into the main branch.

If those steps are successful your new_development branch will be fully integrated within the main branch.

Merging branches with conflicts

However, it is possible that Git will not be able to automatically resolve some conflicts,

```
# Auto-merging index.html
# CONFLICT (content): Merge conflict in index.html
# Automatic merge failed; fix conflicts and then com
```

Important: **DO NOT PANIC**

You will need to resolve the conflicts

- You will have to resolve them **manually**.
- This normally happens when two branches have the same file but with two different versions of the file. In that case Git is not able to figure out which version to use and is asking you to resolve the conflict.

Resolving merging conflicts

- First thing to do --> figure out which files are those affected by the conflict
- git status

```
git status
# On branch main
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
#     both modified:      example.Rmd
#
# no changes added to commit (use "git add" and/or "
```

Resolving the conflict

- Open the file with a text editor
- Go to the lines which are marked with

  <<<<<<, ====== , and >>>>>>

- Edit the file

- git add filename
- git commit -m "Message"
- git push origin main

Resolving conflicts

When you open the conflict file in a **text editor such as Rstudio Cloud / Rstudio**, you will see the conflicted part marked like this:

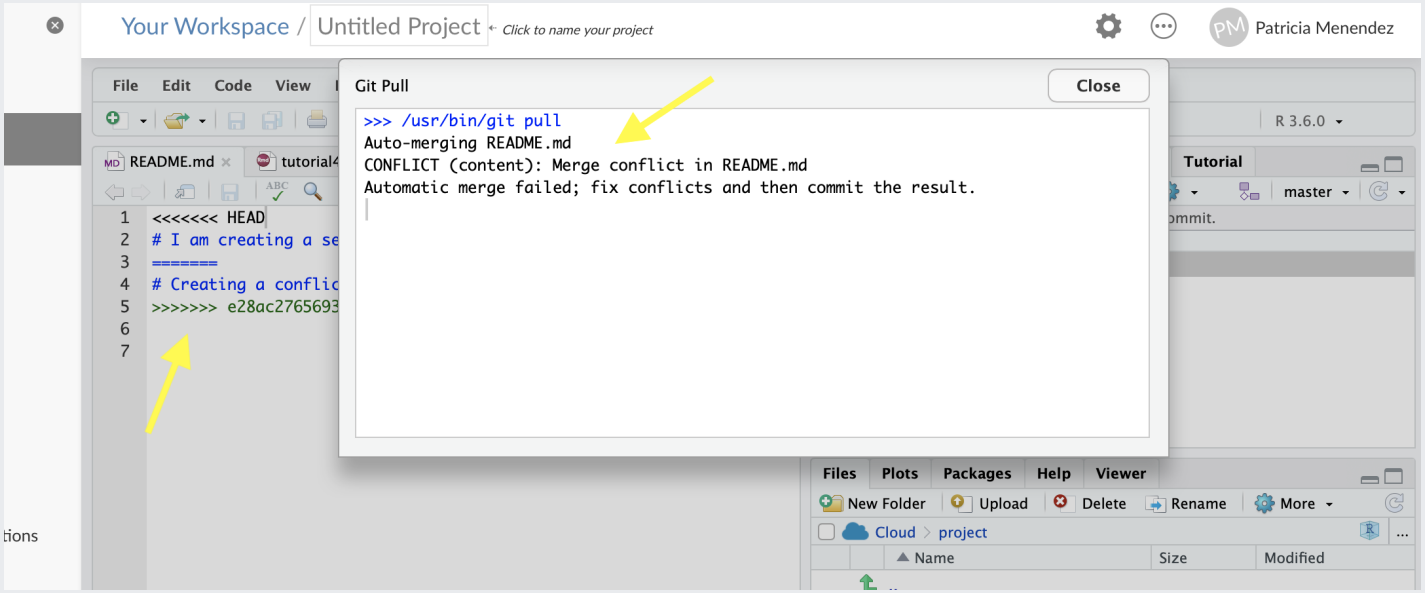
```
/* code unaffected by conflict */
<<<<<<< HEAD
/* code from main that caused conflict */
=======
/* code from feature that caused conflict */
>>>>>>>
```

When Git encounters a conflict, it adds <<<< ; >>>> and ======= to highlight the parts that caused the conflict and need to be resolved.

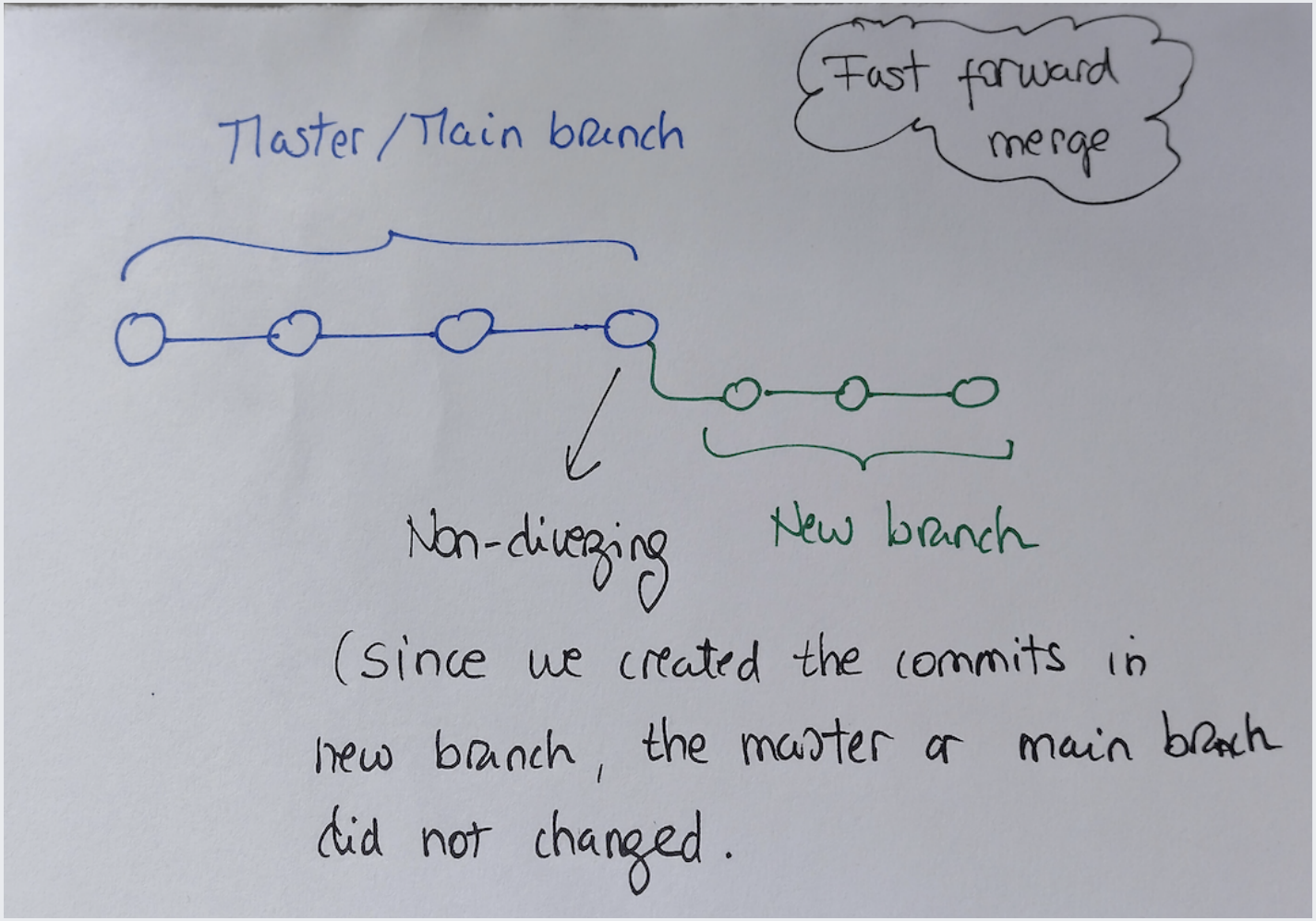
Resolving conflicts in practice

- Open the file in a text editor (for example Rstudio / Rstudio Cloud)
- Decide which part of the code you need to keep in the final main branch
- Removed the irrelevant code and the conflict indicators
- Run git add to stage the file/s and git commit to commit the changes --> this will generate the merge commit.

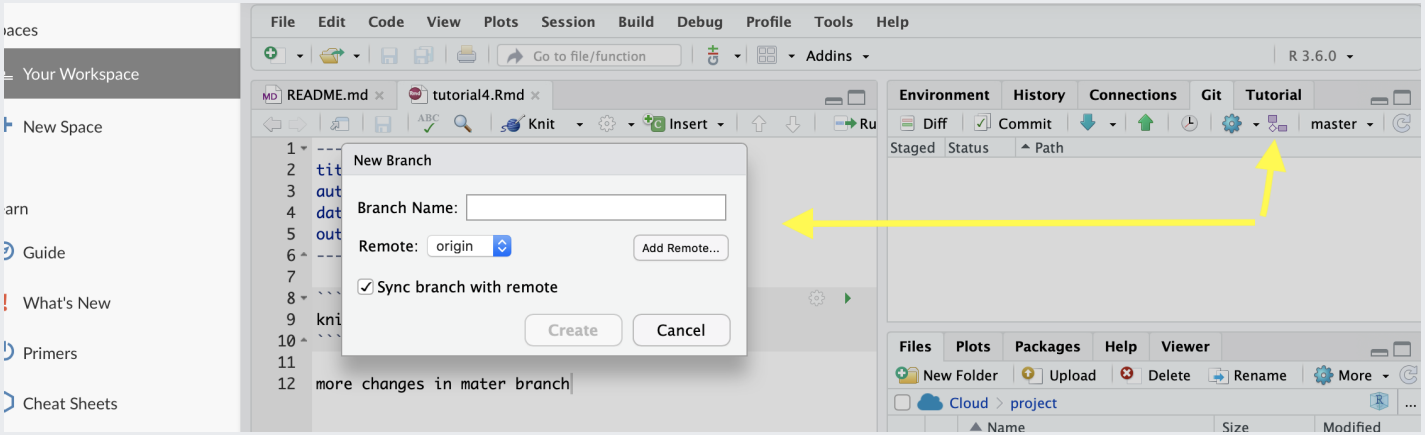
Resolving the conflict



Merging branches



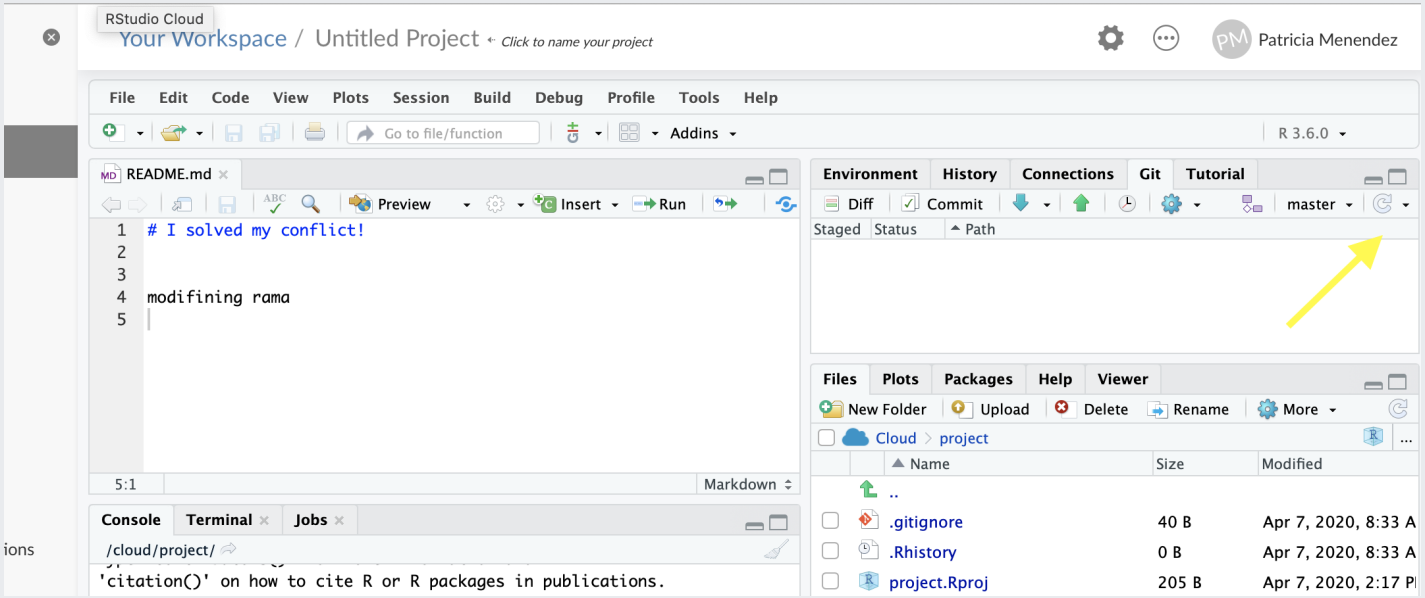
Creating branches from Rstudio



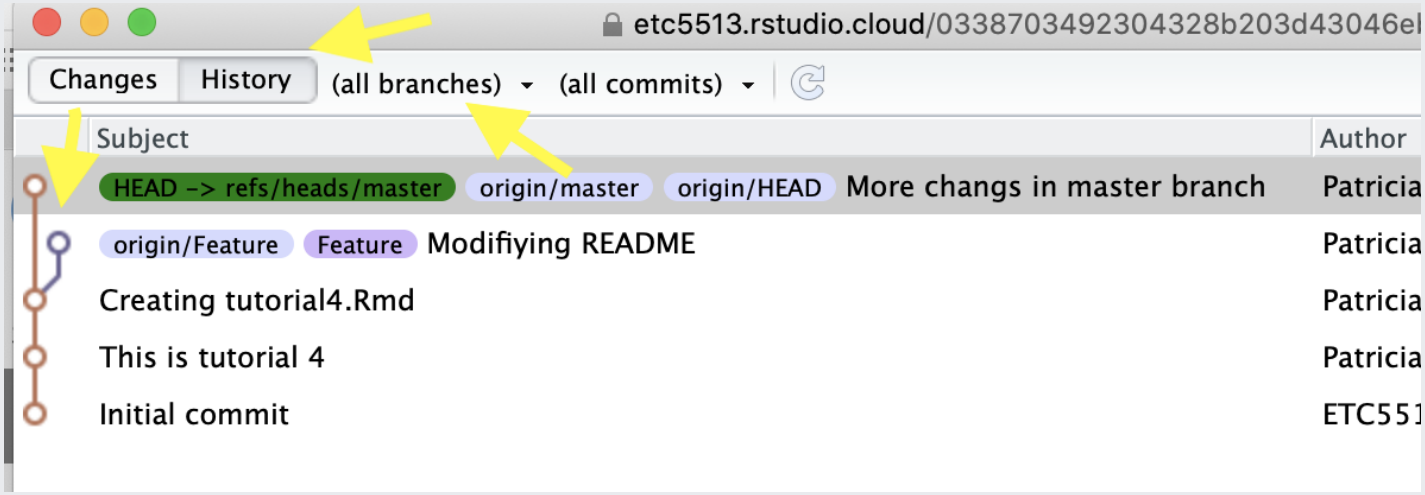
Important: When we create a branch using Rstudio the branch is created both in the local and in the remote repository (at the same time.)

Keep refreshing Rstudio cloud

Otherwise some of your branches and changes might not be updated.



Diff window in Rstudio



Rstudio Demo

Few things to start getting installed

Please follow the link and get the Github Education pack.

- From now on we will be using VS code. Please look at instructions in Week 3 to get it installed.
- Please get Gitkraken (from the github education pack for students)

https://education.github.com/students

Time to start our programs locally!

After your tutorial this week:

In the next weeks, we will be using GitKraken and VS code

- To help us resolve conflicts
- To have a nicer way for visualizing Git trees + much more!
- Once you get your Github Education pack, you will also have access to Gitkraken and you can install VS code as per instructions last week. Make sure you get those installed before your next tutorial.

Lecturer: Patricia Menéndez

Department of Econometrics and Business Statistics