

An Intro to Git and GitHub

A crash course

Dr Patricia Menéndez

Department of Econometrics
and Business Statistics

What is a version control system?

"Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later."

It also allows us to collaborate and share our projects with others!

Book on Git here - fantastic resource.



Version control

- Version control systems are a category of software tools that help store and manage changes to source code (projects) over time.
- Version control software keeps track of every modification to the source code in a special kind of database.
- If a mistake is made, you can turn back to previous versions and compare the code to fix the problem while minimizing disruption.
- It is easy to manage multiple versions of a project.
- It is a very useful (actually essential!) tool for collaborating and for sharing open source resources.

Distributed Version Control: Git

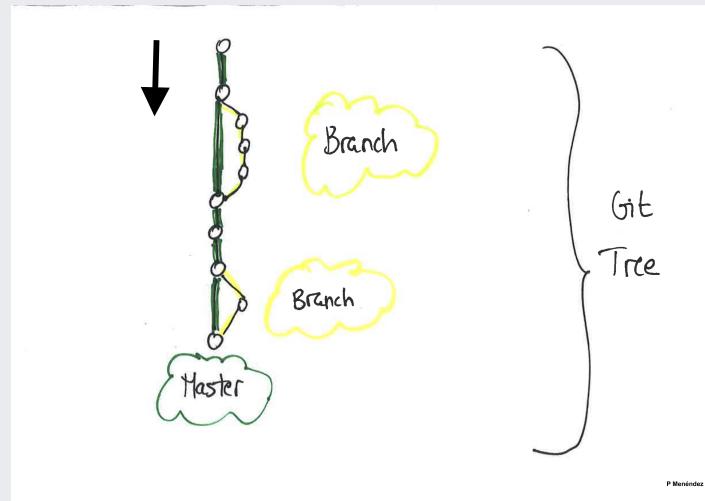
We are going to use a distributed version control called Git



Git overview in a nutshell

Let's think of the connections between the different versions of a project as a tree (Git tree).

- Git tree example.
- White circles represent each version of the project.
- We have what we call **master/main** (default branch).
- We have branches that appeared and then **merged** with the master/main branch.

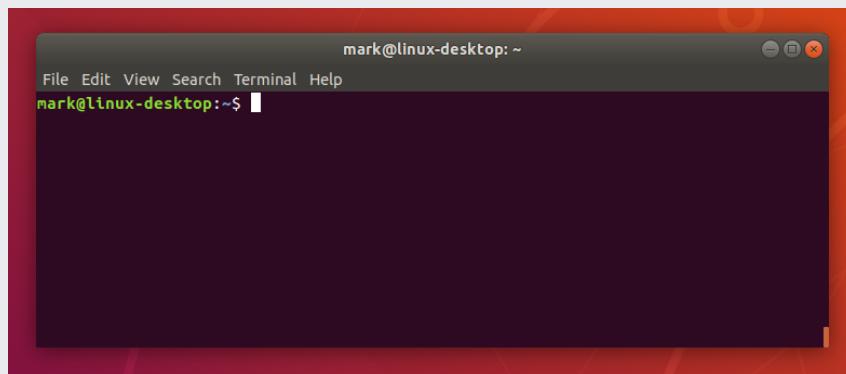


We need to learn

- How Git operates --> shell/command line.
- How to connect our R projects to a Git repo.
- How to connect our local Git repo to a Git Cloud repository (GitHub).

Command Line Interface (cli)

- In most cases (non-linux users) use a Graphical User Interface (GUI) to interact with their programs and operating systems → for example Windows, Mac OX.
- However, at the beginning of the computing times most people would use the command line interface to interact with their computer.



Git and Command Line

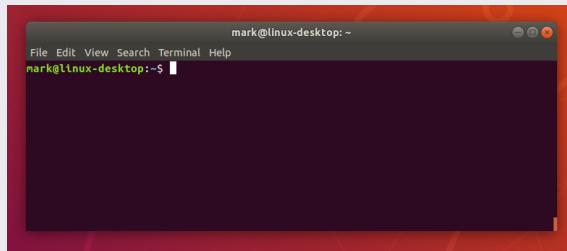
Learn how to use the shell/command line interface.

Why?

- We will use the command line interface to interact with Git and with GitHub.
- The shell or command line interface is an interface where the user types commands.
- This interface allow us to control our computer using commands entered via our keyboard.
- That means that instead of using a graphical user interface (GUI) + our mouse to open and close programs, create folders and moving files --> we are going to type commands.

Command Line Interface

Also known as the Shell, command line interface (cli) or terminal is an interface for **typing** commands to interact directly with a computer's operating system.



Examples of things that we can do from the shell or terminal:

- Navigating through folders and files
- Create/delete folders
- Run and install programs (i.e interact with Git)
- And much more :-)!

Terminal in action

Typically when you open your terminal, it will welcome you with a prompt that looks like this:

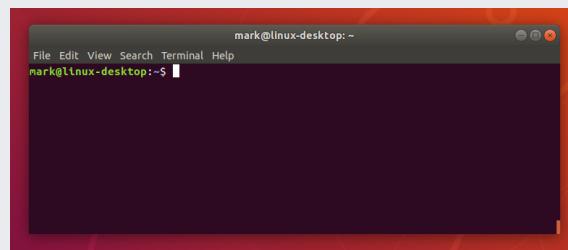
```
patricia@computerid-macbook:~$
```

or with the new Catalina Mac OS

```
patricia@computerid ~ %
```

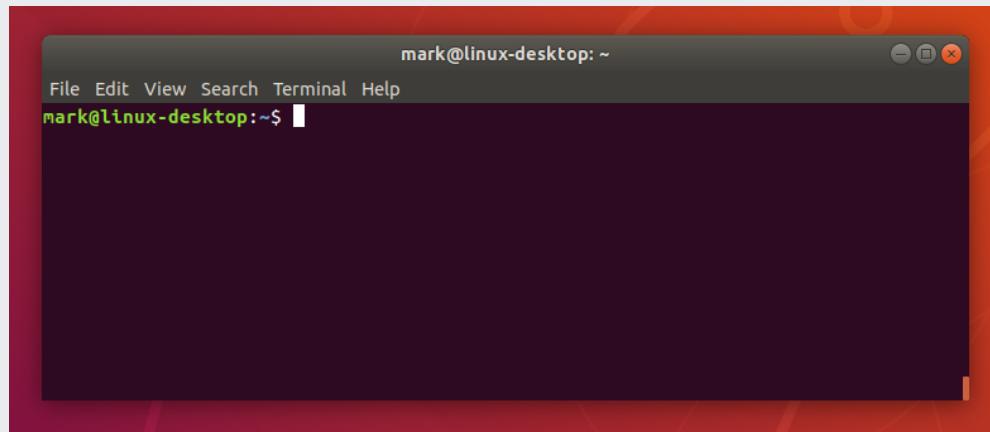
On Windows it will contain the same elements but look like this:

```
patricia@computerid-pc MINGW64 ~$
```



Terminal in practice

We will start writing commands after `~$` or `~%` depending on the terminal version that you are using



- The commands that we are going to use are the same regardless the terminal version you have.

Let's start learning the commands to navigate our computer and to interact with Git via the command line interface



What is the path to my current computer location?

- **pwd** --> print working directory/present working directory

```
patricia@computerid ~ % pwd
```

```
/Users/patricia/Documents/ETC5513
```

Understanding the output of **pwd command:**

```
/Users/patricia/ETC5513/GitHub
```

- / --> represents the root directory
- Users --> is the Users directory
- patricia --> refers to my directory or folder within the users directory
- ...

Contents within the location --> directory

- Listing the contents of the directory or folder where I am:

ls --> list files inside my directory

```
patricia@computerid Documents~ % ls
```

```
Courses Research Teaching file.pdf example.txt
```

- **Documents** is an argument to the **ls** command.
- **ls** → gives you a list of all the elements in a directory
- **ls -a** → list of **all** the files including **hidden ones**.

Each Linux command (pwd,ls ...) have lots of options (**flags**) that can be added.

A reference list of unix commands with options might be found here.

Command Line Basics: Navigating between directories

cd --> change directory

- First we need to make sure where we are **pwd**. We can also use **pwd** to figure how we can get where we want to go!
- The **cd** command syntax is very simple --> we just need to specify the directory that we want to navigate to.
- At any moment of your navigation, you can use the **pwd** command to confirm your current location.
- A path that starts with / is assumed to be absolute.

Cheat sheet for command line

Excellent summary about the commands that we will be using
can be found [here](#)

You don't need to learn all the commands only those that we
are going to use!

Let's get some practice
using the terminal

Demo

Files in a Git repository

The states in a Git repository are: the working directory, the staging area and the git directory:

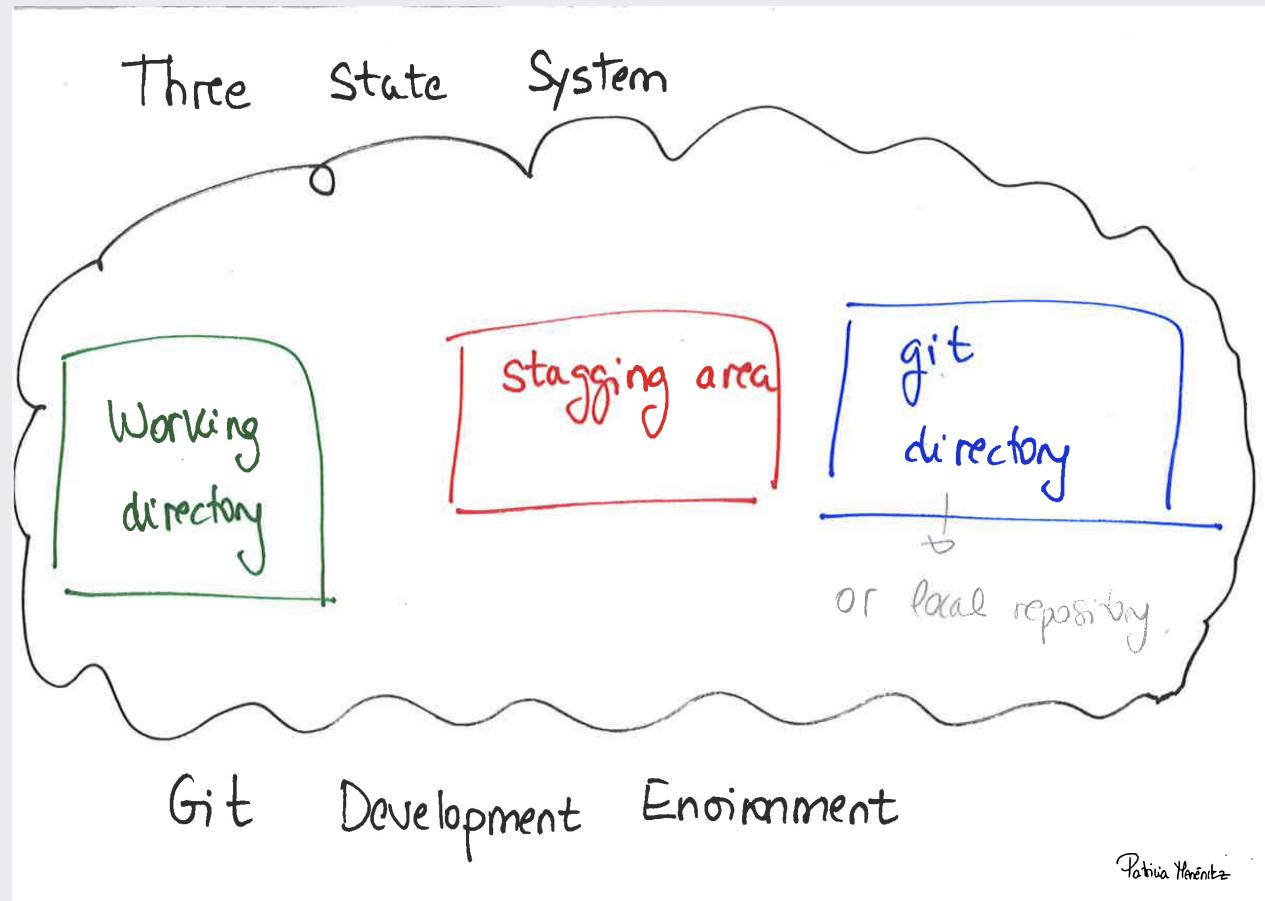
- The **working directory** is the current snapshot that you are working on.
- The **staging area** is where modified files are marked in their current version ready to be stored in the database → index of changes.
- The **git directory** is the database where the history is stored.

In your file system you will see the folder of your project

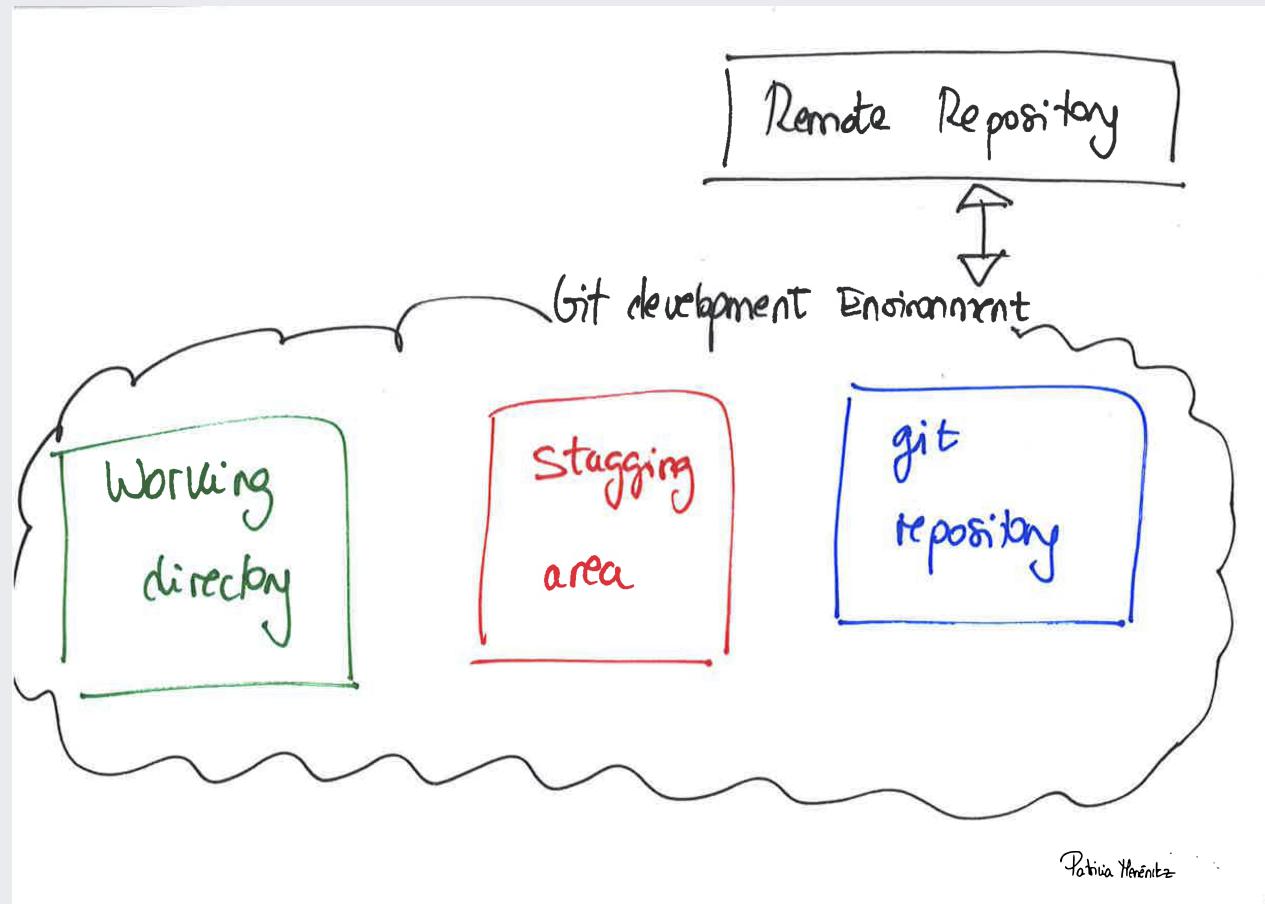
Three state system ~ Git Development Environment

More info here [Begining Git and Github](#)

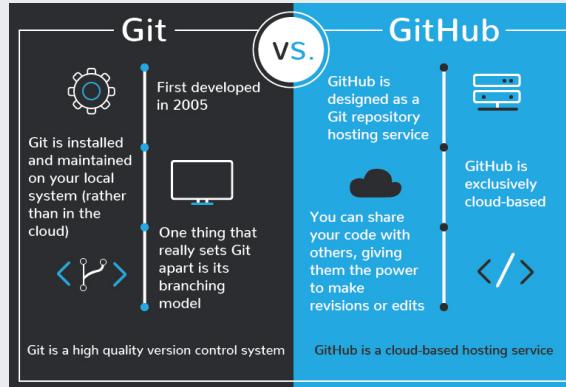
Components of a Git repository: Visually



Git repo and remote repository



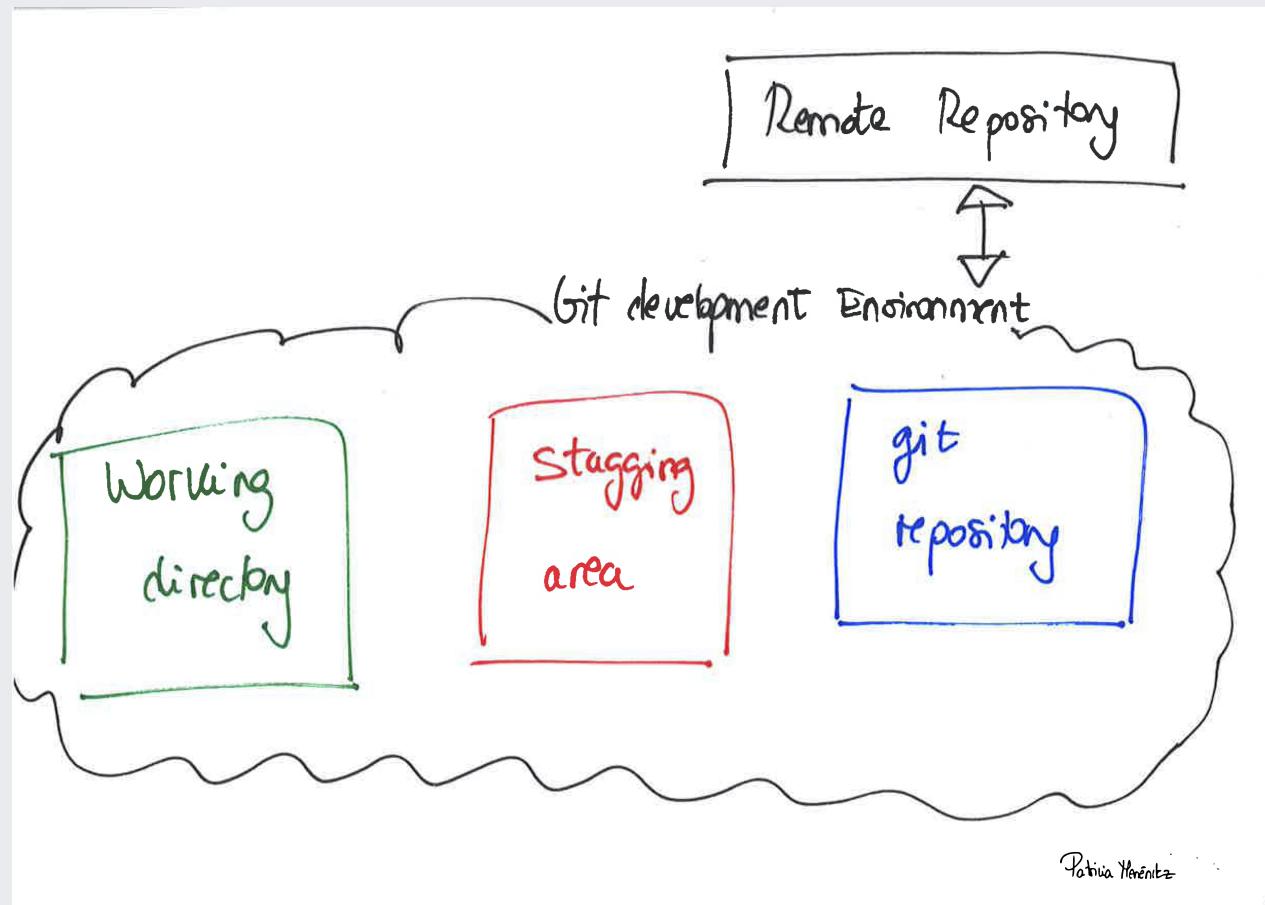
GitHub is our remote repository



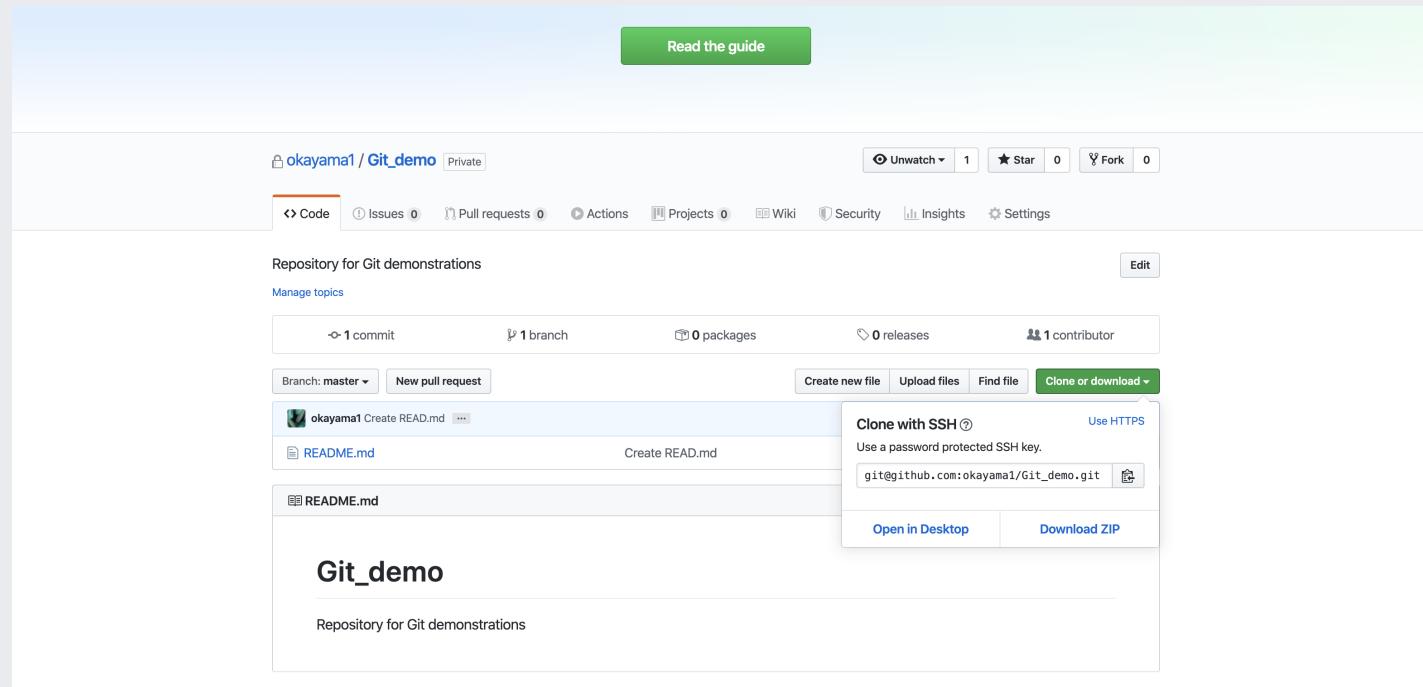
- GitHub is an interface and cloud hosting service built on top of the Git version control system.
- Git does the version control and GitHub stores the data remotely.
- GitHub makes your projects accessible on a fully feature project website.

More info about GitHub here

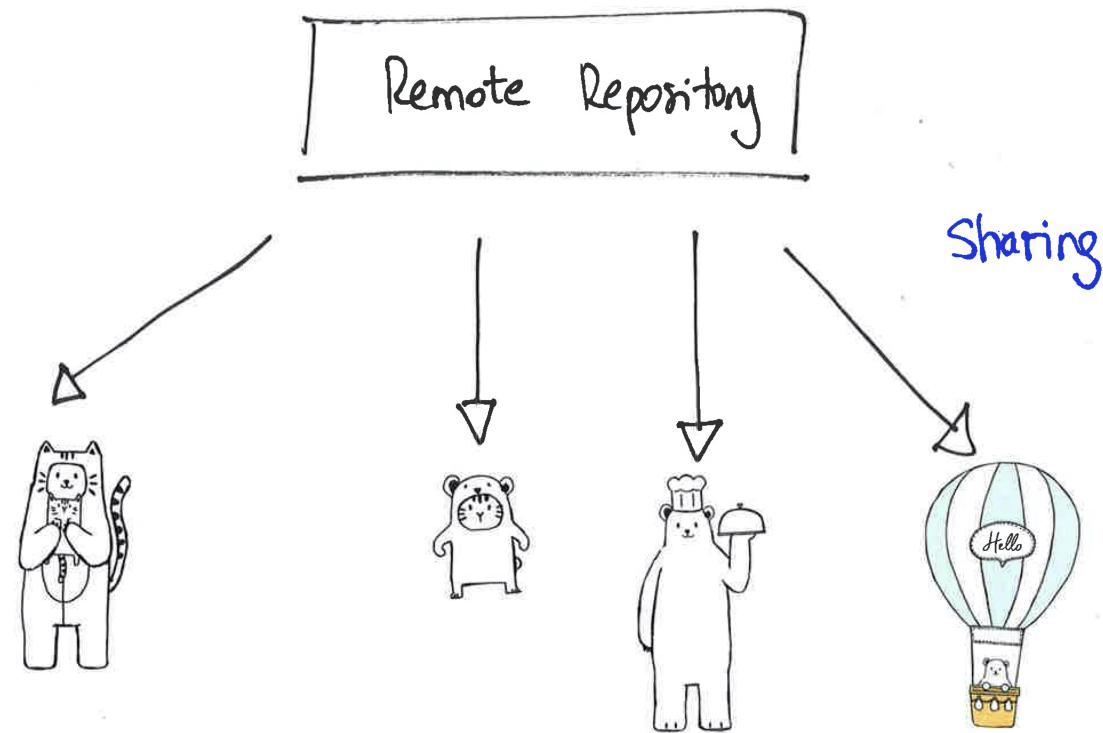
Git repo and a remote repository



GitHub is our remote repository



Collaborative



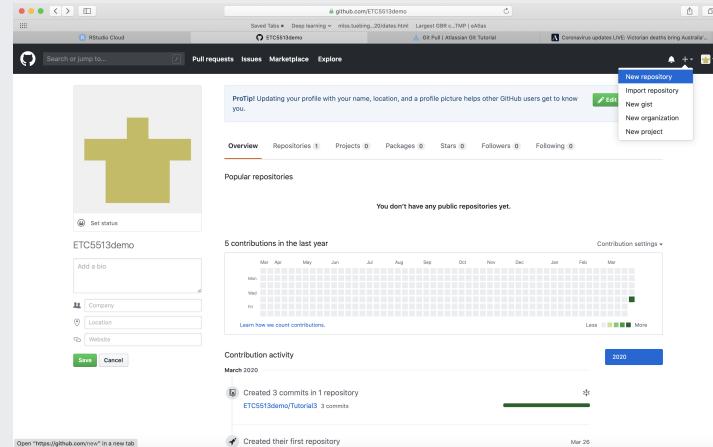
From GitHub <--> to our computer

- Create a Git repository (**repo**) on <https://www.github.com>
- Clone this GitHub repository into our computer → **making a 'local copy'.**
- Work on our local copy of the repo.
- **Stage and Commit** changes to local repository.
- Push those changes into the remote repo in GitHub.

Let's look at all these in more detail!

Creating a repo on GitHub

1. Login into GitHub
2. Click the '+' icon on the top right on the menu bar and select 'New Repository'.



Important:

- Repo name
- Repo --> public or private
- Make sure it is initialized with **REAME.md**: It is important to have a **README.md file** for every repository. GitHub will use this file as the "presentation" of the repository and should briefly describe what the repo is about.

Creating a repo on GitHub

The screenshot shows the GitHub interface for creating a new repository. At the top, there's a dark header bar with the GitHub logo, a search bar, and navigation links for Pull requests, Issues, Marketplace, and Explore. On the right side of the header are icons for notifications, a plus sign, and a gear.

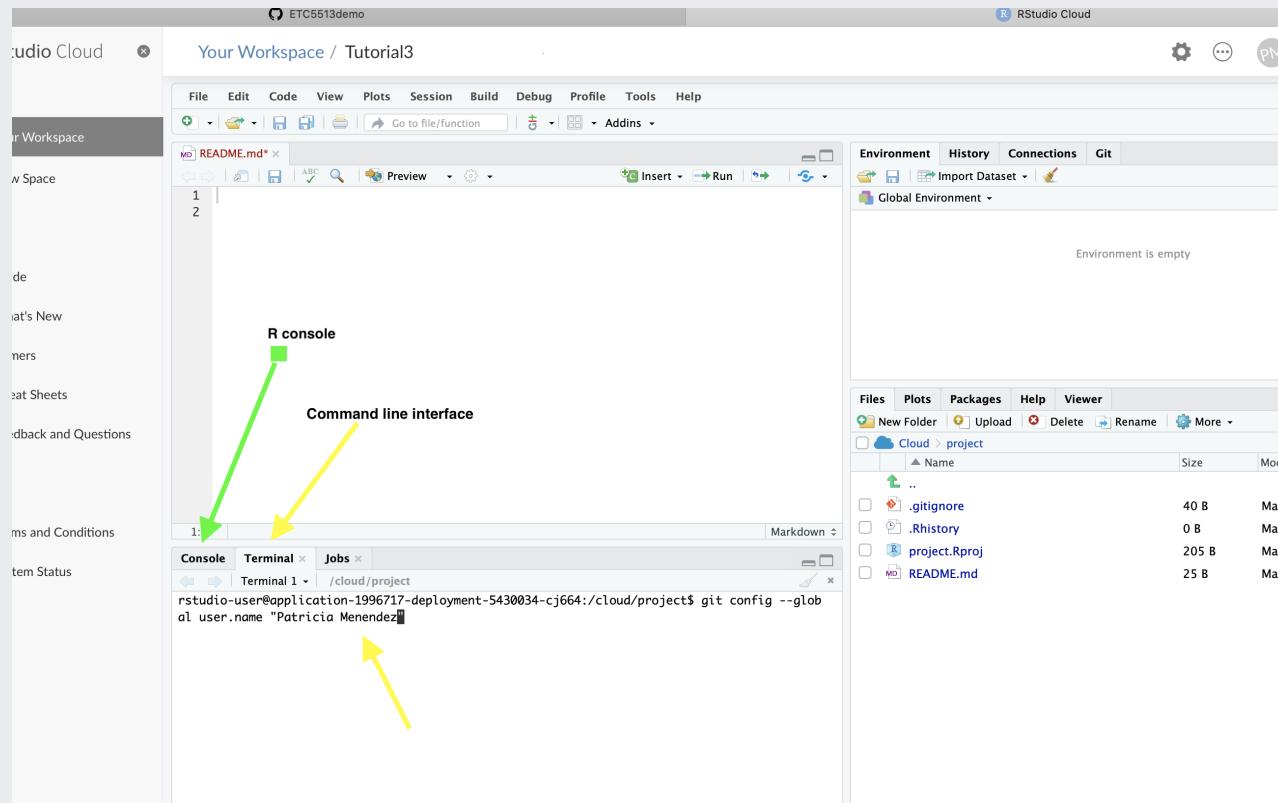
The main content area has a title "Create a new repository". Below it, a sub-instruction says: "A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#)".

The form fields include:

- Owner:** A dropdown menu showing "ETC5513demo" with a yellow icon. To its right is a separator line and a text input field containing "Tutorial3" with a green checkmark icon.
- Description (optional):** A text input field containing "Learning Git".
- Visibility:** Two radio button options:
 - Public**: "Anyone can see this repository. You choose who can commit."
 - Private**: "You choose who can see and commit to this repository."
- Skip this step if you're importing an existing repository.**
- Initialize this repository with a README**: A checked checkbox with the explanatory text "This will let you immediately clone the repository to your computer."
- Add .gitignore:** A dropdown menu set to "None".
- Add a license:** A dropdown menu set to "None".

At the bottom is a large green button labeled "Create repository".

Configuring Git in your Rstudio project using the terminal



Configuring Git in your Rstudio Cloud project

First of all we need to get your Git configured in Rstudio Cloud (the same follows for your own computer):

- Open the command line interface/terminal interface (CLI) and type:
- `git config --global user.email "your.email@example.com"`
- `git config --global user.name "Your_Firstname_Lastname"`

Make sure you use the same email address for this and for setting up your GitHub account.

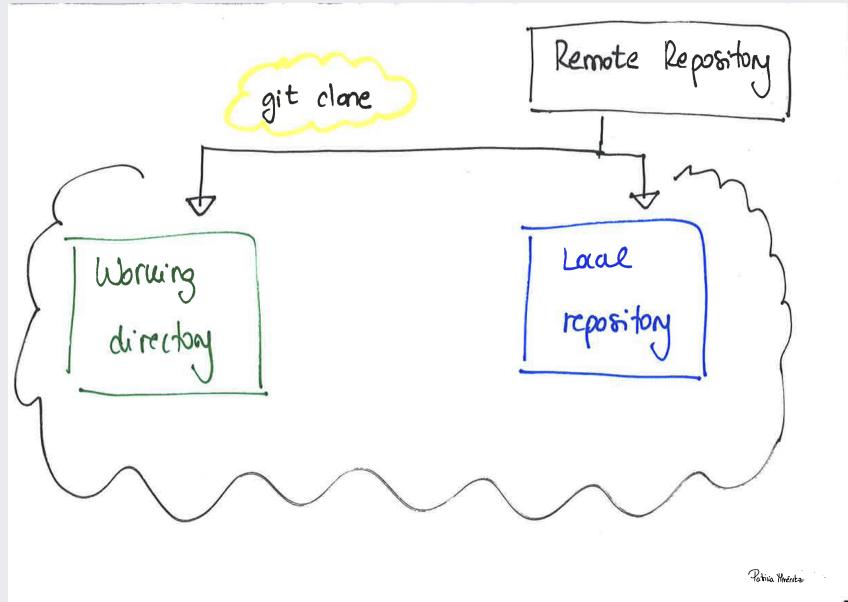
- To check that everything is set up correctly, type the following in the CLI: `git config --global user.email`

Configuring Git in your Rstudio Cloud project using the R console

Alternative you could type the following in your **R console** **inside Rstudio**:

```
usethis::use_git_config(user.name = "Your Name Surname",  
user.email = "Your monash email")
```

Getting a remote repository

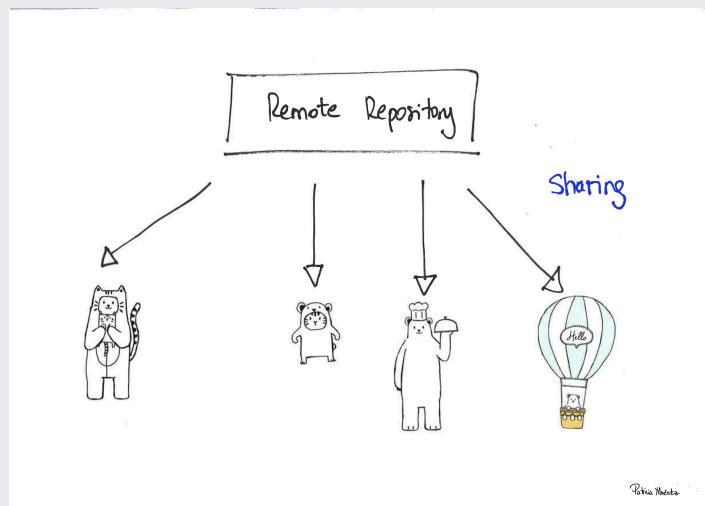


- Grabs remote repository from a server/cloud (i.e GitHub)
- Creates a new folder (copy of the remote repository) in our computer

Cloning a github repo

When you create a Git repository on GitHub, it exists as a remote repository.

Users can clone your repository to create a local copy on their own computer and sync between the two locations.



More detailed info here

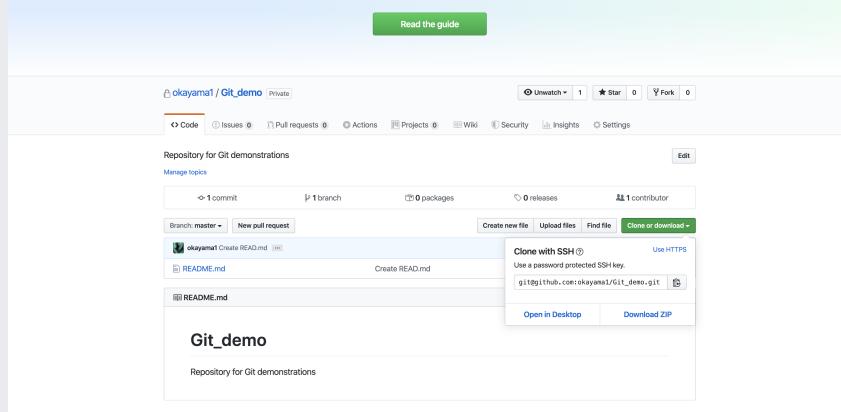
Adding an SHH key into your GitHub

It is very useful to add an SSH key into your workflow:

- When working with a Git repository you will be required to identify yourself to GitHub using your username and password *each time to do a commit!*.
- An SSH key is an alternative way to identify yourself that **does not require you to enter you username and password every time.**
- **SSH keys come in pairs**, a **public key** that gets shared with services like GitHub, and a **private key** that is stored only on your computer or in your Rstudio Cloud project.
- If the keys match, you're granted access!

More info [here](#) and [here](#)

From GitHub to our computer



From our shell or command line:

1. Navigate to the computer location where we want to download the GitHub repo.
2. **git clone git@github.com:okayama1/Git_demo.git**.
3. This will create a folder in your computer with the GitHub repository files and folders.

Three Git States

Git has three main states that your files can reside in:
modified, staged, and committed:

- **Modified** --> you have changed the file but have not committed it to your repository database yet.
- **Staged** --> you have marked a modified file in its current version to go into your next commit snapshot.
- **Committed** --> the data is safely stored in your local database.

This leads us to the three main sections of a Git project: the working tree, the staging area, and the Git directory.

Three Git States Visually

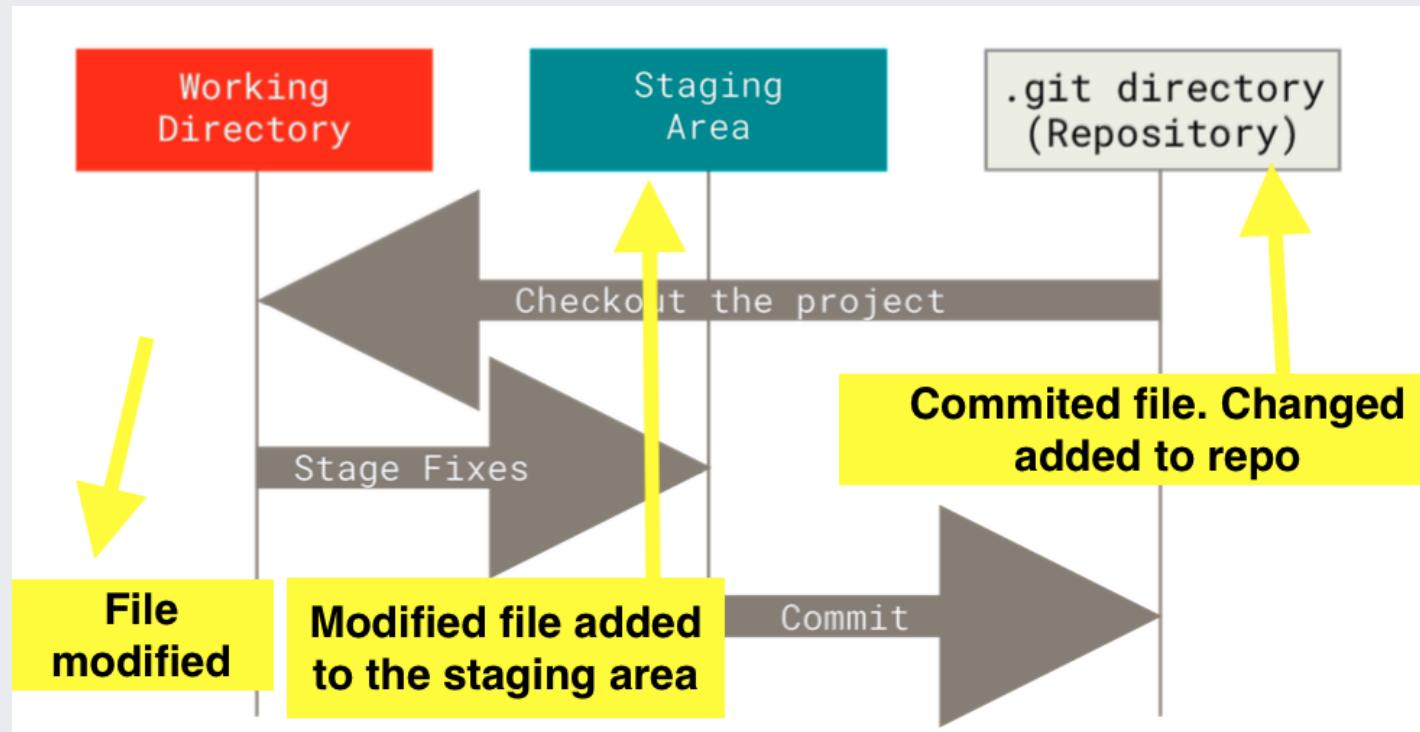
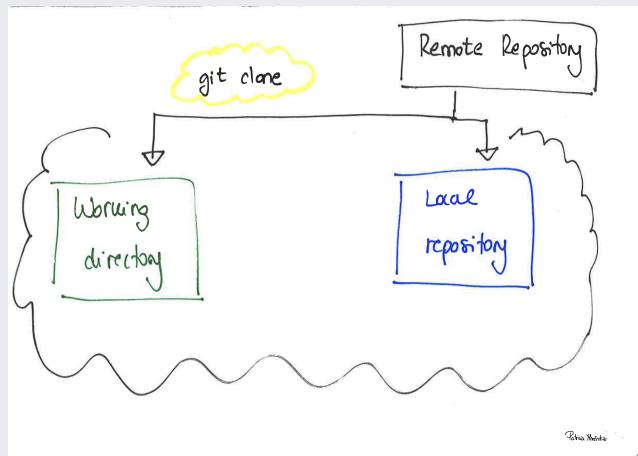
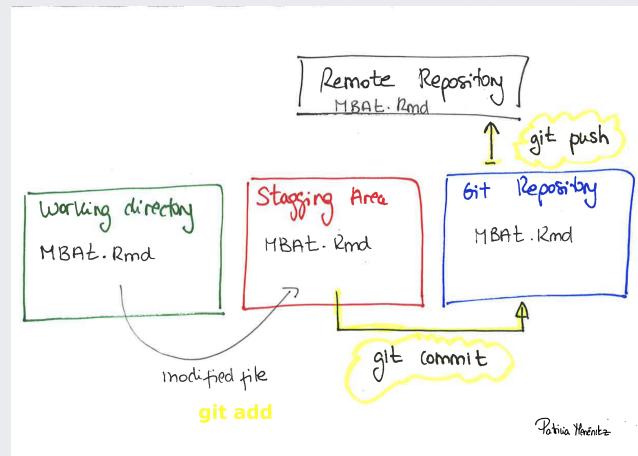


Figure source

GitHub Workflow: Visual example



clone info --> GitHub repo



Working in your computer and updating remote repo in GitHub

Tracked and untracked files

- In a git repository tracked files are those which are part of the git repository
- However, we can also have untracked files for which their history is not tracked
- Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about.
- Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area.

More info [here](#).

From git clone to first commit

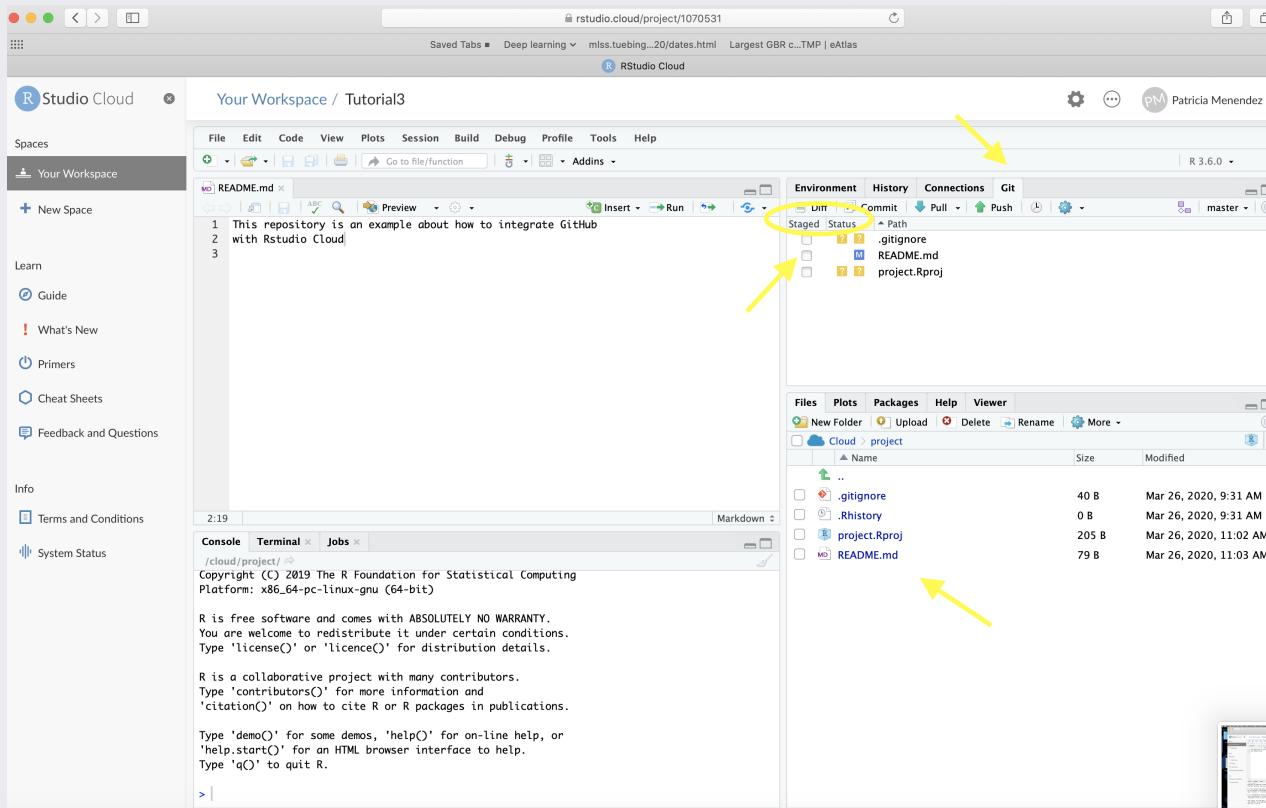
- `git clone "remote repo address"` → is a Git command used to target an existing repository and create a clone (or copy) of the target repository.
- `git add filename` → it adds a change in the working directory to the staging area.
- `git commit -m "Message"` The git commit command captures a snapshot of the project's currently staged changes. (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 main` → The git push command is used to upload local repository content to a remote repository, in this case to the main branch.

General Workflow (via Terminal)

- `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.

Rstudio and GitHub

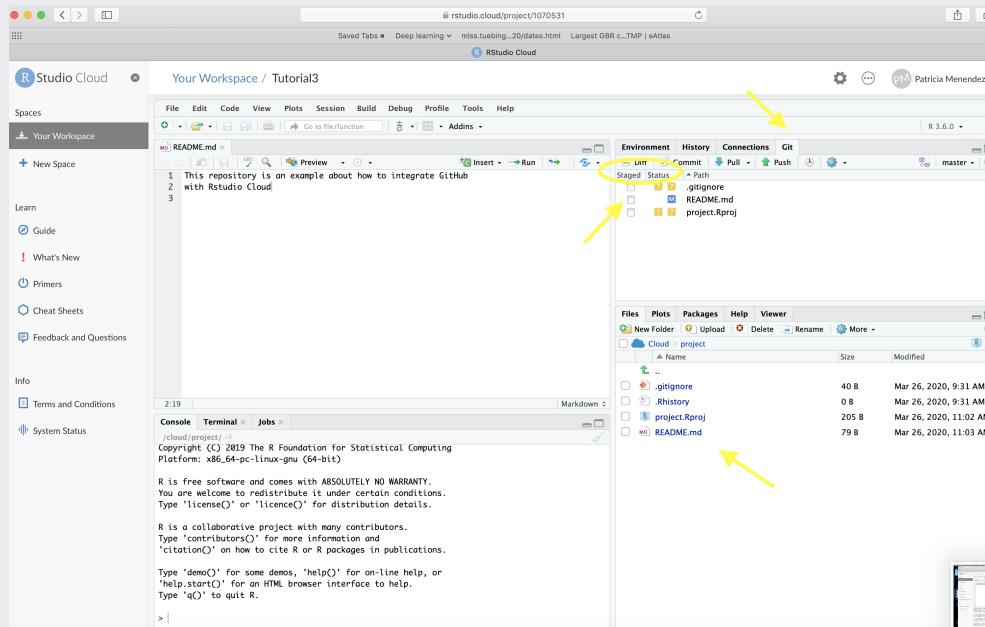
The status/staging panel in Rstudio



Rstudio and GitHub

RStudio keeps Git constantly scanning the project directory to find any files that have changed or which are new.

- By clicking a file's little "check-box" you can stage it.



Rstudio and GitHub

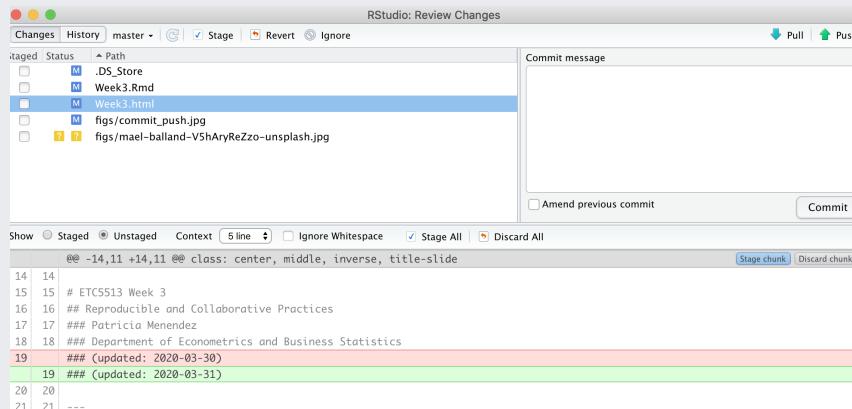
Understanding the symbols in the Rstudio Git pane:

- **Blue-M**: a file that is already under version control that has been modified.
- **Orange-?**: a file that is not under version control (yet...)
- **Green-A**: a file that was not under version control, but which has been staged to be committed.
- **Red-D**: a file under version control has been deleted. To make it really disappear, you have to stage its disappearance and commit.
- **Purple-R**: a file that was renamed. (Note that git in Rstudio seems to be figuring this out on its own.)

Changes

The Diff window

- Shows what has changed between the last committed version of a file and its current state.
- Note: all this output is available from the command line too, but the Rstudio interface is very nice!



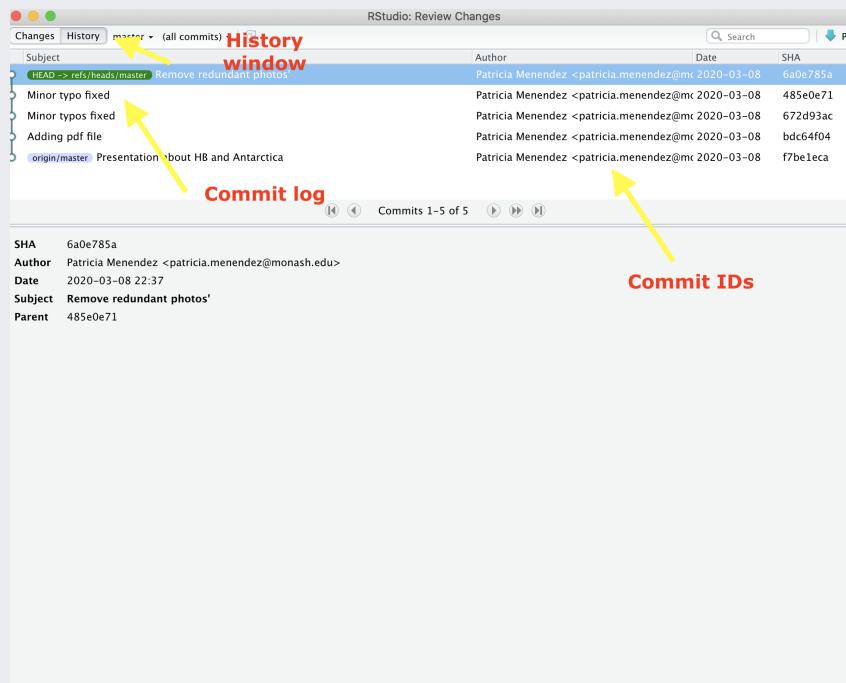
The screenshot shows the RStudio 'Review Changes' window. The left pane lists files with their status: .DS_Store (M), Week3.Rmd (M), Week3.html (M), figs/commit_push.jpg (M), and figs/mael-balland-VShAryReZzo-unsplash.jpg (D). The 'Week3.html' file is selected. The right pane contains a 'Commit message' field and a 'Commit' button. Below the message field is an 'Amend previous commit' checkbox. At the bottom, there are buttons for 'Stage chunk' and 'Discard chunk'. The bottom section displays a diff of the 'Week3.html' file:

```
@@ -14,11 +14,11 @@ class: center, middle, inverse, title-slide
14 14
15 15 # ETC5513 Week 3
16 16 ## Reproducible and Collaborative Practices
17 17 ### Patricia Menendez
18 18 ### Department of Econometrics and Business Statistics
19 19 ### (updated: 2020-03-30)
20 20 ### (updated: 2020-03-31)
21 21 ---
```

The History window

Allow us to understand past commits.

- Easy inspection past commits.
- See what changes were made at each commit.



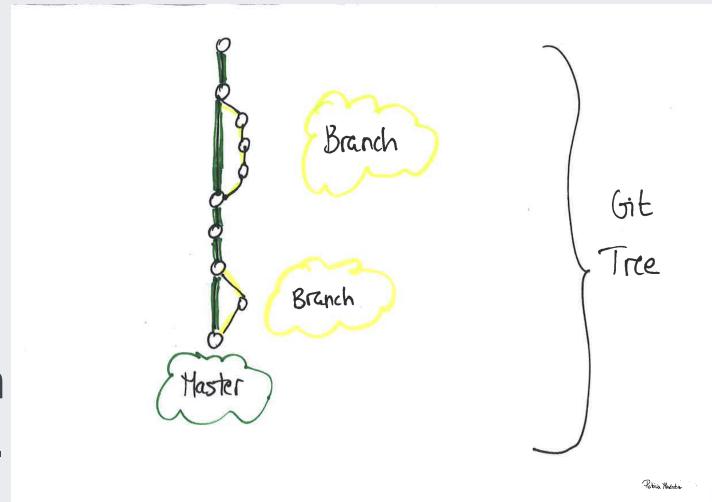
Important:

- We can interact between Git, GitHub and our local repository using the terminal only.
- We can interact between Git, GitHub and our local repository using Rstudio.

Branching

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

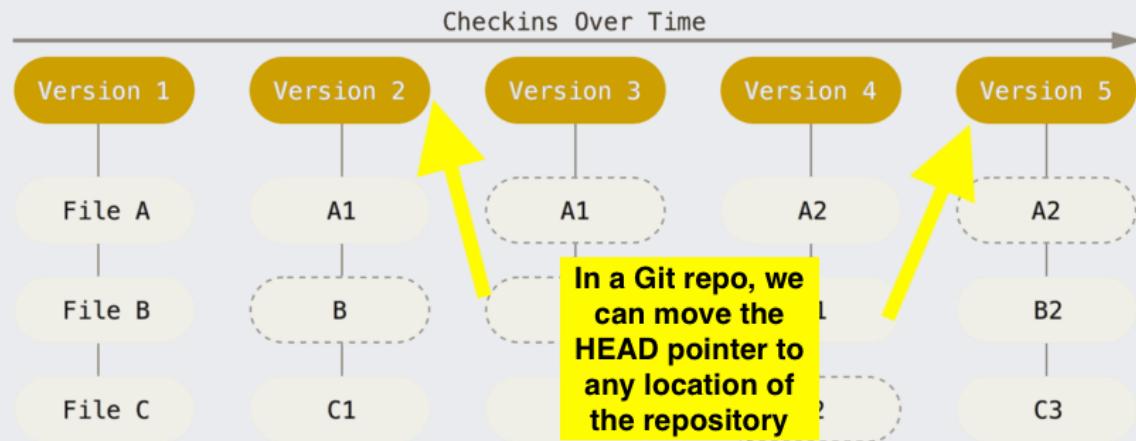
- It allows you to duplicate your existing repository
- Use a branch to isolate development work without affecting other branches in the repository
- Modification in a branch can be merged into your project.



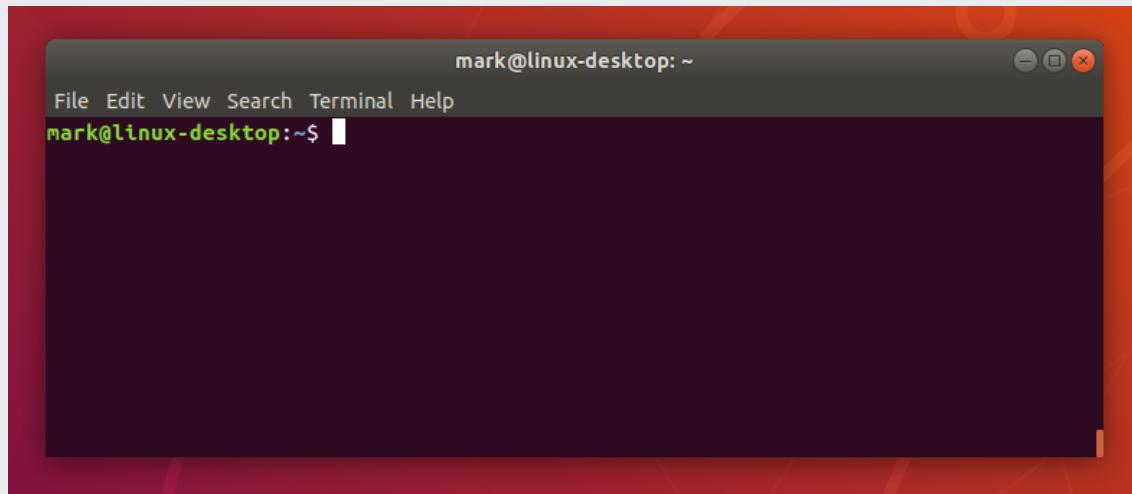
Branching is particularly important with Git as it is the mechanism that is used when you are collaborating with other researchers/data scientists.

HEAD

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







For the time, we will be using the Terminal within Rstudio.
However, I personally prefer to use the external terminal.

Create branches using the Terminal/Shell/CLI

Using `git branch` and `git checkout` commands

- `git branch` --> shows 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.

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`

Alternatively 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

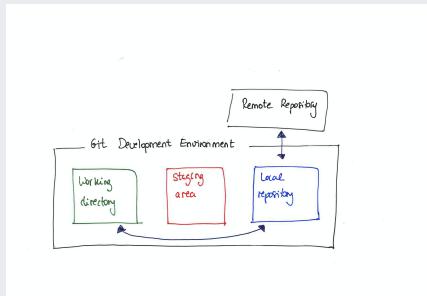


Deleting branches using cli

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`

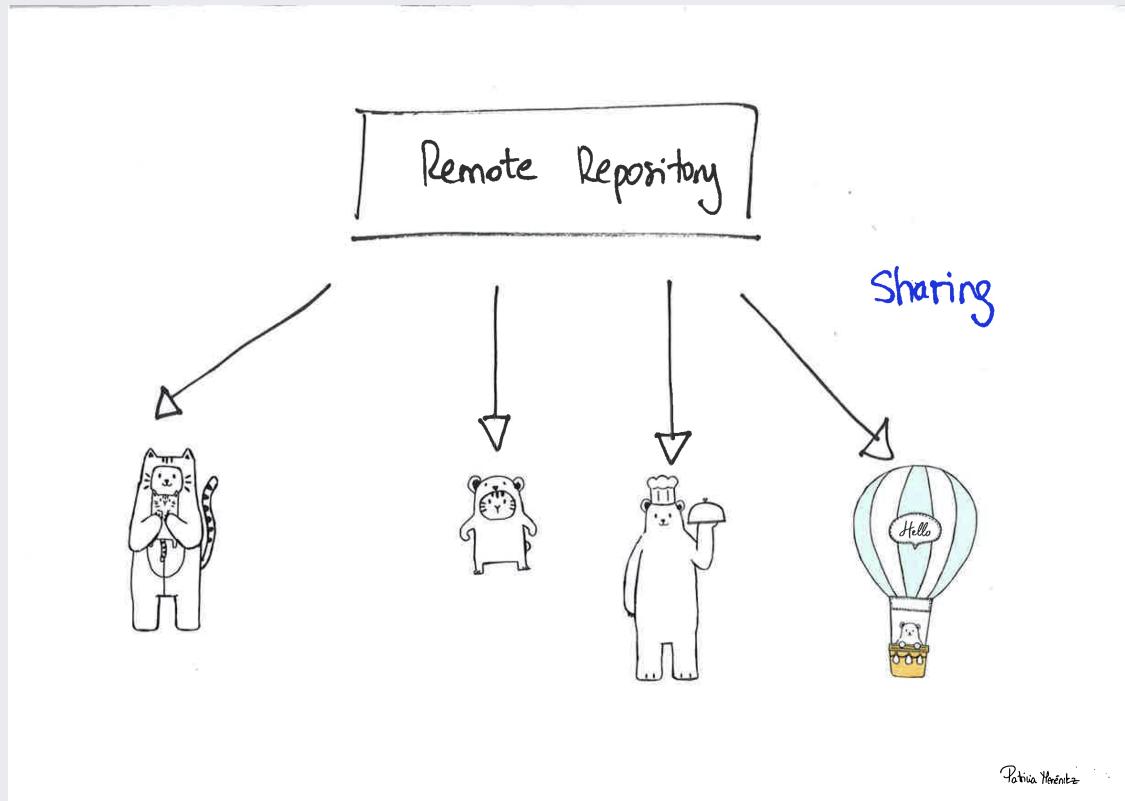
You cannot delete a branch if your HEAD is on that branch



Deleting branches using cli

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

1. `git push origin --delete Name_of_branch`



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` --> Alternative_analysis
- To go back to main (assuming that you were in there) `git checkout main`

Merging branches successfully

Suppose we have two branches: `main` and `new_development`

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

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 the same file but with two different versions. In that case Git is not able to figure out which version to use.

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 "git rm"
```

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:

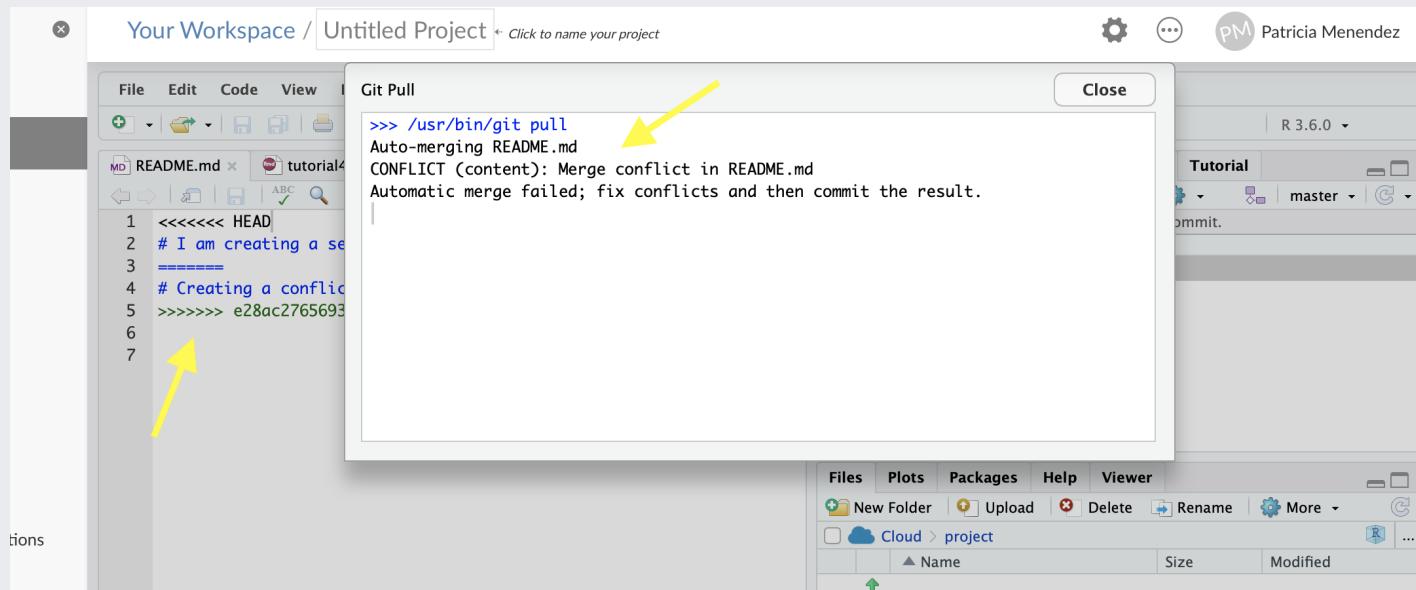
```
/* code unaffected by conflict */  
=<<<<< HEAD  
/* code from main that caused conflict */  
=====  
/* code from feature that caused conflict */
```

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

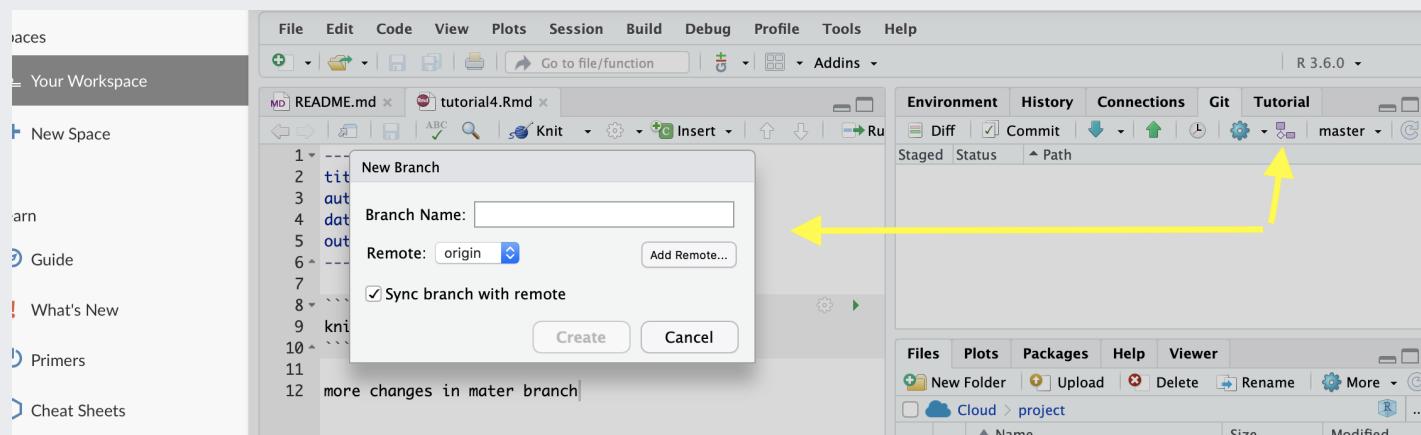
Resolving conflicts in practice

- 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



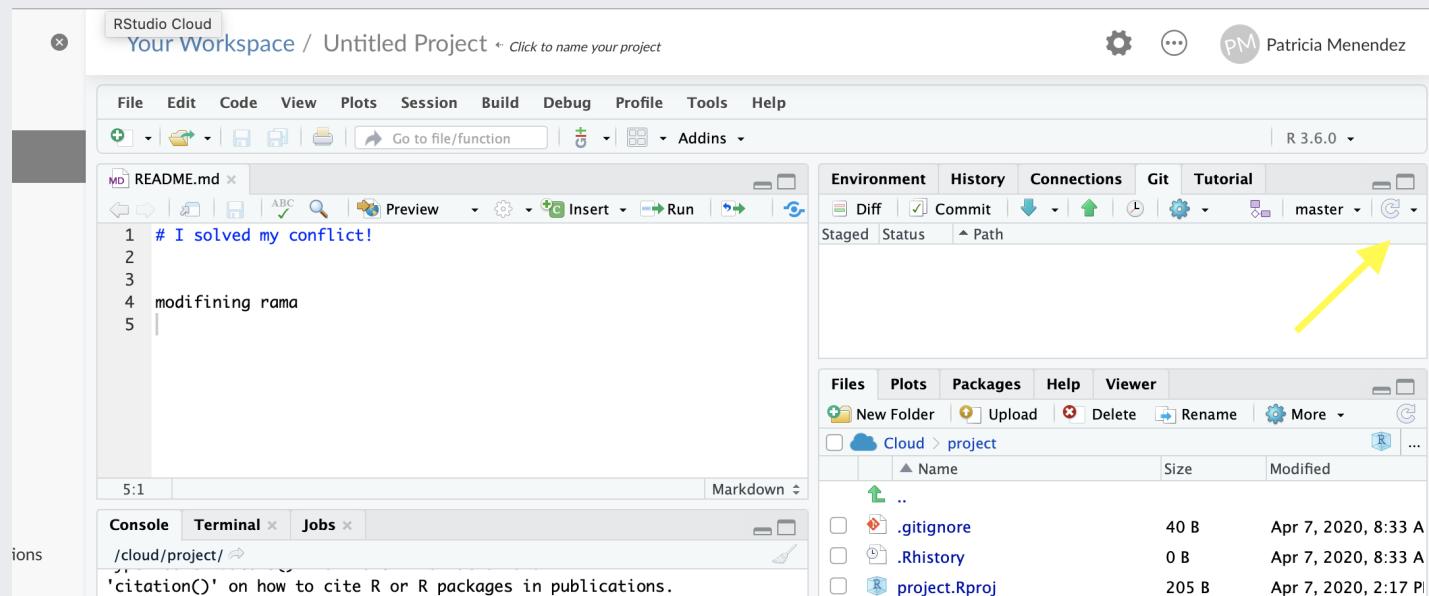
Creating branches from Rstudio



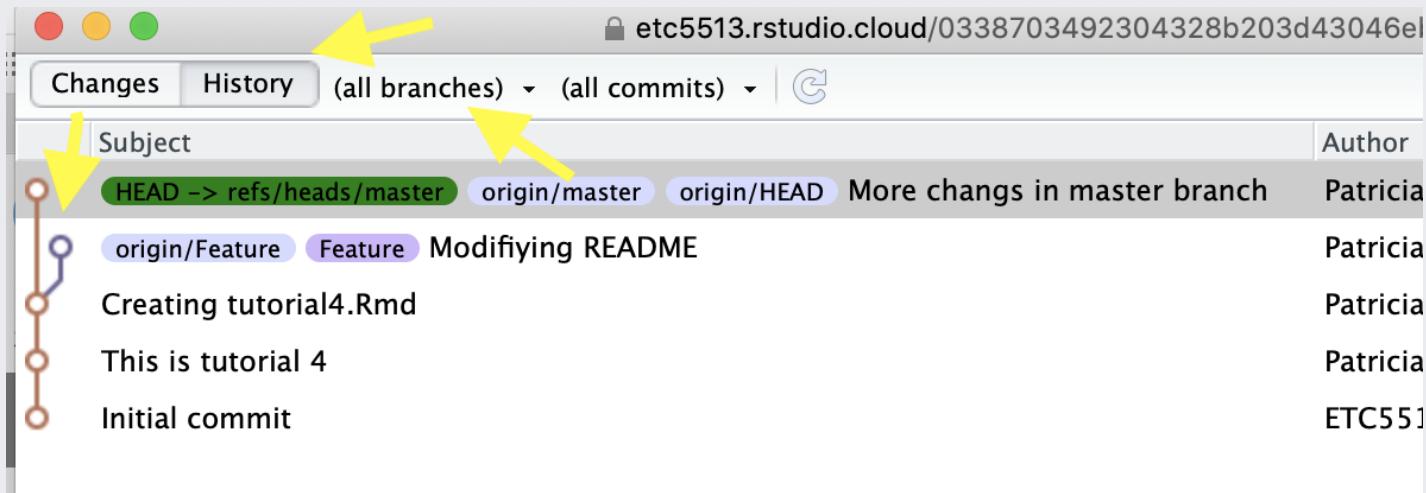
Important: When we create a branch using Rstudio the branch is created both in the local and in the remote repository (at the same time.)

Keep refreshing Rstudio/Rstudio cloud

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



Diff window in Rstudio



Few things to get that are worth

Please follow the link below and get the GitHub Education pack.
Then you can use:

- VS code (Install VS code for your commits and)
- Use VS code to visualize your Git Tree

<https://education.github.com/students>

Three important topics

- GitHub issues --> Great for collaboration. Please see read more [here](#).
- Licenses --> Choose the right License
- .gitignore --> See more [here](#)

Lecturer: Dr Patricia Menéndez

Department of Econometrics and Business Statistics

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

