Introduction to Data Science

Session 2: Version control and project management

Simon Munzert
Hertie School

Table of contents

[1] Much of this lecture draws on materials from Grant McDermott's excellent *Data Science for Economists* class.

Project management

Taming chaos

In the data science workflow, there are two sorts of **surprises** and cognitive stress:

1. Analytical (often good)
2. Infrastructural (almost always bad)

Analytical surprise is when you learn something from or about the data.

Infrastructural surprise is when you discover that:

- You can't find what you did before.
- The analysis code breaks.
- The report doesn't compile.
- The collaborator can't run your code.

Good project management lets you focus on the right kind of stress.

Keeping Future-you happy

- It’s often tempting to set up a project assuming that you will be the only person working on it, e.g. as homework.
- That’s almost never true.
- Coauthors and collaborators happen to the best of us.
- Even if not, there's someone else who you always have to keep happy: Future-you.
- Future-you is really the one you organize your projects for.
- They are who you use version control for (see later).
- Most importantly, they are who will enjoy the fruits of your data science labor, or have to fight back your chaos.
- So, be kind to Future-you. Establish a good workflow. You'll thank yourself later.

Keeping Future-you happy

- It’s often tempting to set up a project assuming that you will be the only person working on it, e.g. as homework.
- That’s almost never true.
- Coauthors and collaborators happen to the best of us.
- Even if not, there's someone else who you always have to keep happy: Future-you.
- Future-you is really the one you organize your projects for.
- They are who you use version control for (see later).
- Most importantly, they are who will enjoy the fruits of your data science labor, or have to fight back your chaos.
- So, be kind to Future-you. Establish a good workflow. You'll thank yourself later.



Project setup

You should **always** think in terms of projects.

A project is a **self-contained unit of data science work** that can be

- Shared
- Recreated by others
- Packaged
- Dumped

Project setup

You should **always** think in terms of projects.

A project is a **self-contained unit of data science work** that can be

- Shared
- Recreated by others
- Packaged
- Dumped

A project contains

- Content, e.g., raw data, processed data, scripts, functions, documents and other output
- Metadata, e.g., information about tools for running it (required libraries, compilers), version history

Project setup

You should **always** think in terms of projects.

A project is a **self-contained unit of data science work** that can be

- Shared
- Recreated by others
- Packaged
- Dumped

A project contains

- Content, e.g., raw data, processed data, scripts, functions, documents and other output
- Metadata, e.g., information about tools for running it (required libraries, compilers), version history

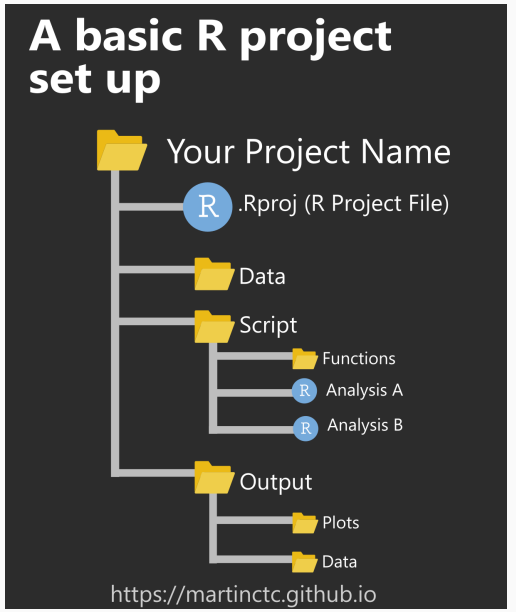
For R projects

- Projects are folders/directories.
- Metadata is the RStudio project (`.Rproj`) files (perhaps augmented with the output of renv for dependency management) and `.git`.

Setup: the folder structure

Structuring your working directory

- One folder contains everything inside it.
- Directories keep things separate that should be separated.
- You decide on the fundamental structure. The project decides on the details.



Further thoughts

- Ideally, your project folder can be relocated without problem.
- Keep input separate from output. Definitely separate raw from processed data!
- Structure should be capable of evolution. More data, cases, models, output formats shouldn't be a problem.

```
.
└── my_awesome_project
    ├── src
    ├── output
    ├── data
    │   ├── raw
    │   └── processed
    ├── README.md
    ├── run_analyses.R
    └── .gitignore
```

`Credit` Chris/r-bloggers.com

Setup: the paths

Good paths

- All internal paths are relative.
- They are invariant to moving/sharing the project.
- Examples:
  - `"preprocessing.R"`
  - `"figures/model-1.png"`
  - `"../data/survey.RDa"`

Bad paths

- Using `setwd()` is bad practice 99% of the times.
- Absolute paths are bad paths. Don't feed functions with paths like `"/Users/me/data/thing.sav"`.
- Those paths will not work outside your computer (or maybe not even there, some days/weeks/months ahead).

Setup: the paths

Good paths

- All internal paths are relative.
- They are invariant to moving/sharing the project.
- Examples:
  - `"preprocessing.R"`
  - `"figures/model-1.png"`
  - `"../data/survey.RDa"`

Bad paths

- Using `setwd()` is bad practice 99% of the times.
- Absolute paths are bad paths. Don't feed functions with paths like `"/Users/me/data/thing.sav"`.
- Those paths will not work outside your computer (or maybe not even there, some days/weeks/months ahead).

The working directory

- Set it manually once per session (do `Session > Set Working Directory > Choose Directory`). Then all your good paths will "just work".
- Better yet, get it right automatically by opening RStudio with clicking on the script you want to work with. This will set the location of the script as working directory (which should be your working assumption, too).
- Even better yet, have the metadata set it for you:
  - Open your session by opening (choosing, clicking on) `myproject.Rproj`
  - Then you'll get the path set for you.
- That's probably better than the previous option because you might not want your `code` directory to be the working directory.

Setup: the code structure

Naming scripts

- Files should have short, descriptive names that indicate their purpose.
- I recommend the use of telling verbs.
- Names should only include letters and numbers with dashes `-` or underscores `_` to separate words.
- Use numbering to indicate the order in which files should be run:
  - `0-setup.R`
  - `1-import-data.R`
  - `2-preprocess-data.R`
  - `3-describe-uptake.R`
  - `4-analyze-uptake.R`
  - `5-analyze-experiment.R`

Setup: the code structure

Naming scripts

- Files should have short, descriptive names that indicate their purpose.
- I recommend the use of telling verbs.
- Names should only include letters and numbers with dashes `-` or underscores `_` to separate words.
- Use numbering to indicate the order in which files should be run:
  - `0-setup.R`
  - `1-import-data.R`
  - `2-preprocess-data.R`
  - `3-describe-uptake.R`
  - `4-analyze-uptake.R`
  - `5-analyze-experiment.R`

Modularizing scripts

- Write short, modular scripts. Every script serves a purpose in your pipeline.
- This makes things easier to debug.
- At the beginning of a script you might want to document input and output.

Setup: the code structure (cont.)

Talk to Future-you

- Describe your code, e.g. by starting with a description of what it does. If you comment/describe a lot, consider using an R Markdown (`.Rmd`) file instead of a simple `.R` script.
- Put the setup first (e.g., `library()` and `source()`).
- You might want to outsource the loading of packages to a separate script that is imported in the first step (`source("functions.R")`) or just declared the first script in the pipeline.
- Always comment more than you usually do.

Setup: the code structure (cont.)

Talk to Future-you

- Describe your code, e.g. by starting with a description of what it does. If you comment/describe a lot, consider using an R Markdown (`.Rmd`) file instead of a simple `.R` script.
- Put the setup first (e.g., `library()` and `source()`).
- You might want to outsource the loading of packages to a separate script that is imported in the first step (`source("functions.R")`) or just declared the first script in the pipeline.
- Always comment more than you usually do.

Structuring your code

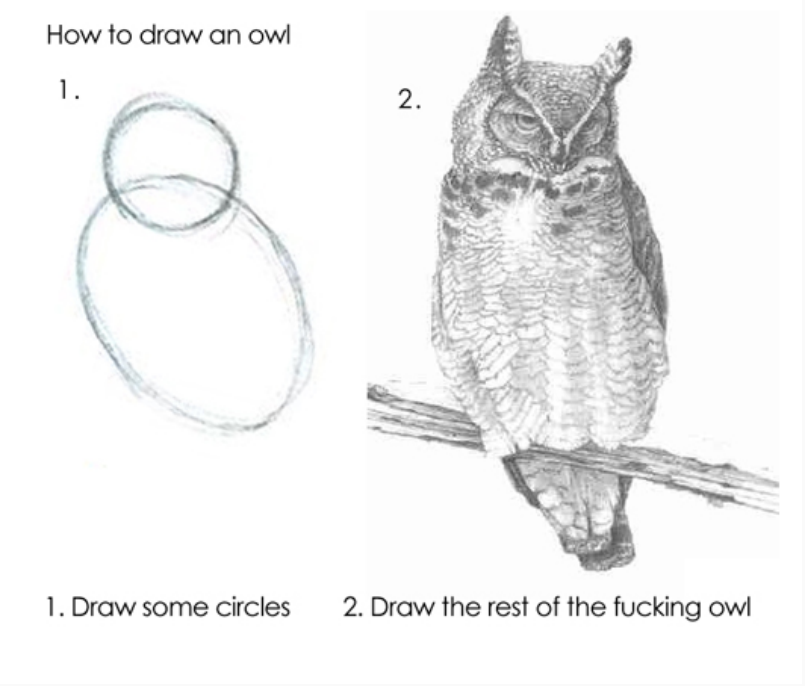
- Even with modularized code, scripts can become long. Structure helps to keep an overview.
- Use commented lines as section/subsection heads.
- RStudio creates a "table of contents" when you name your code chunks as follows (`#` followed by title and `---`):

```
R> # Import data --------------
R>
R> dat ← read_csv("dat.csv")
```

Setup: the rest

More things to consider

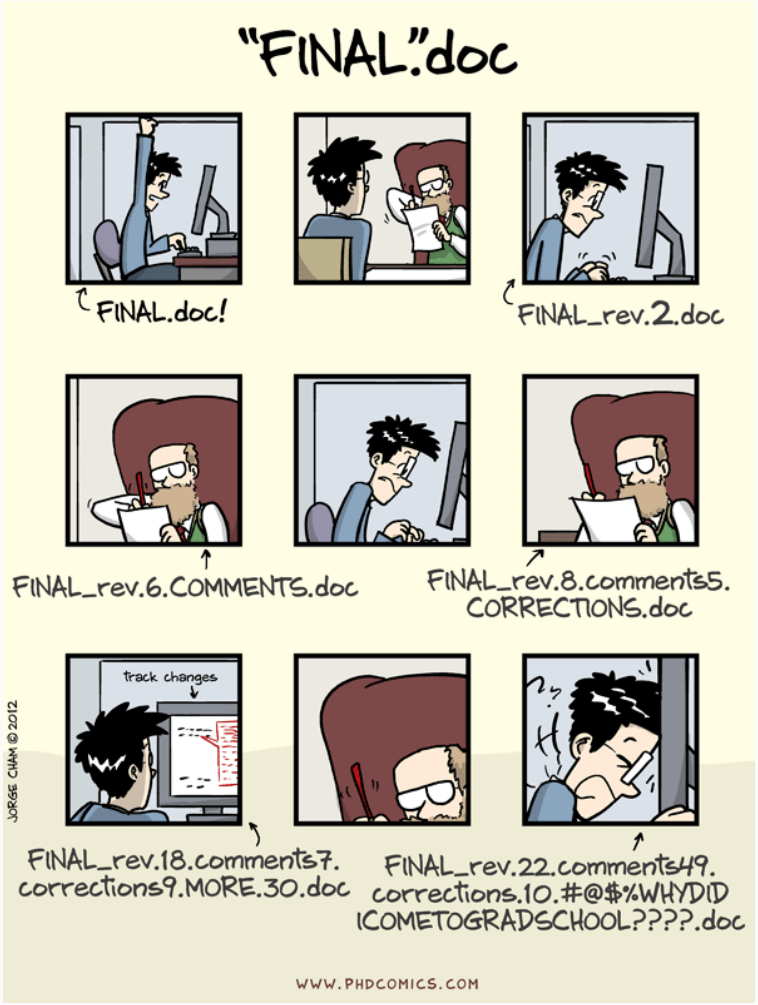
- There'd be more to say on how to establish a good project workflow, including how to
  - store/organize raw and derived data,
  - deal with output in form of graphs and tables,
  - link everything together from start (project setup) to finish (knitting the report)
  - separate coding for the record and experimental coding.
- But there's limited value in teaching you all that upfront.
- The truth is: You'll likely refine your own workflow over time. I just saved you some initial pain (hopefully).
- Do check out other people's experiences and opinions, e.g., here or here or here.



Managing your R project in two simple steps

Git and GitHub

Why version control?



`Credit` phdcomics.com

More reasons to do version control

Have you ever...

- Changed your code, realized it was a mistake and wanted to revert back?
- Lost code or had a backup that was too old?
- Wanted to see the difference between different versions of your code?
- Wanted to review the history of some code?
- Wanted to submit a change to someone else's code?
- Wanted to share your code, or let other people work on your code?
- Wanted to see how much work is being done, when, and by whom?
- Wanted to experiment with but not interfering with working code?

`Credit` si618/Stackoverflow

`Credit` bhimanshukalra/devrant.com

Git(Hub) solves this problem

- Git is a distributed version control system.
- Imagine if your Dropbox (or Google Drive, or MS OneDrive for that matter) and the "Track changes" feature in MS Word had a baby.
- In fact, it's even better than that because Git is optimized for the things that data scientists spend a lot of time working on - code!
- There is a learning curve, but it's worth it.
- Being familiar with Git is taken for granted when you interact with other data scientists.
- It is by far not the only version control software, but certainly the most popular one.
- According to StackOverflow's 2021 Developer Survey, more than 93% of respondents report to use Git - more than any other tool.

Git(Hub) solves this problem

- Git is a distributed version control system.
- Imagine if your Dropbox (or Google Drive, or MS OneDrive for that matter) and the "Track changes" feature in MS Word had a baby.
- In fact, it's even better than that because Git is optimized for the things that data scientists spend a lot of time working on - code!
- There is a learning curve, but it's worth it.
- Being familiar with Git is taken for granted when you interact with other data scientists.
- It is by far not the only version control software, but certainly the most popular one.
- According to StackOverflow's 2021 Developer Survey, more than 93% of respondents report to use Git - more than any other tool.

- It's important to realize that Git and GitHub are distinct things.
- GitHub is an online hosting platform that allows you to host your code online.
- It relies on Git and makes some of its functionality more accessible.
- Also, it provides many more useful features to collaborate with others. (Similar platforms include Bitbucket and GitLab.)
- Just like we don't *need* Rstudio to run R code, we don't *need* GitHub to use Git... But it will make our lives easier.

Git: some background

Where does Git come from?

- Git was created in 2005 by Linux creator Linus Torvalds.
- The initial motivation was to have a non-proprietary version control system to manage Linux kernel development.
- Check out this (quite opinionated) talk by Linus Torvalds on Git two years after its creation.

What's the meaning of `Git`?

- Anything, apparently.
- Also, it's pronounced [gɪt], not [d͡ʒɪt].

How to interact with Git?

- There are many Git GUIs, giving you the option to use git without the shell (often with reduced functionality). A popular choice is GitHub Desktop, but we will mainly use the Git integration into the RStudio IDE.

`Credit` Krd (photo), Von Sprat (crop/extraction), CC BY-SA 4.0.

GitHub: some background

Where does GitHub come from?

- GitHub.com launched in April 2008 by Tom Preston-Werner, Chris Wanstrath, P.J. Hyett and Scott Chacon.
- In 2018, Microsoft acquired the company for more than US$7 billion.

What's the business model?

- GitHub offers various subscription plans and has expanded its services beyond hosting Git-based version control.

Some interesting facts

- GitHub's mascot is "Octocat", a human-cat-octopus hybrid with five arms.
- There are 56m+ developers on Github, with 60m+ new repositories created in 2020 alone.
- Part of GitHub's history are controversies around issues like harassment allegations or incidences of censorship.

`Credit` Screenshot

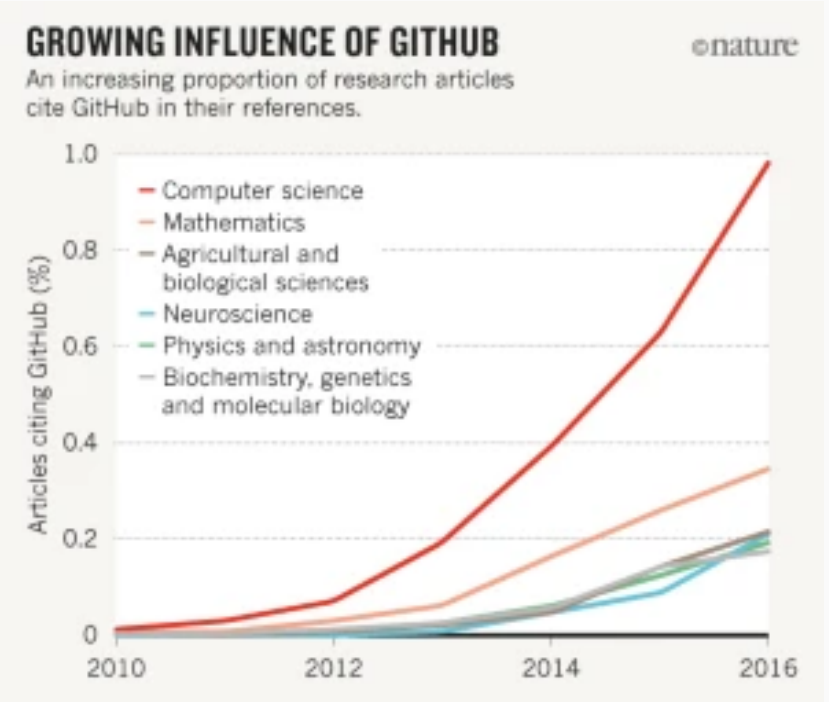
Git(Hub) for scientific research

From software development...

- Git and GitHub's role in global software development is not in question.
- There's a high probability that your favourite app, program or package is built using Git-based tools. (RStudio is a case in point.)

... to scientific research

- Data science involves product building, collaboration, transparency. GH helps with all that.
- Journals have increasingly strict requirements regarding reproducibility and access. GH makes this easy (DOI integration, off-the-shelf licenses, etc.).
- I host most of the code and data for my papers on GH. My website lives there. And this course does, too.



`Credit` "Democratic databases: science on GitHub" (Perkel, 2016, *Nature*).

Getting started

First step: register a GitHub account

Good news: It's free!

Simply go to https://github.com to sign up.

Some things to consider:

- As a student, you qualify for a free GitHub Pro account.
- The Pro account comes with a couple of additional features.
- Register for a free account first, then pursue the special offers.
- Choose your username wisely. This isn't Instagram, so maybe avoid puns and "funny" nicknames.

Second step: install Git

Again, Git is an independent piece of software. You need to have it installed on your machine to call it from RStudio or the command line.

Chances are that that's already the case. Here's how you can check using the command line:

```
$ which git
```

```
## /usr/bin/git
```

And here's how you can check the version:

```
$ git --version
```

```
## git version 2.32.1 (Apple Git-133)
```

If you want to install (or update) Git on your Mac/Linux machine, I recommend using Homebrew, "the missing package manager for macOS (or Linux)":

```
$ brew install git
```

To install/update Git for Windows, check out happygitwithr.com.

Third step: introduce yourself to Git

This is particularly important when you work with Git but without the GitHub overhead. The idea is to define how your commits are labelled. Others should easily identify your commits as coming from you.

Have you already introduced yourself to Git? Find it out:

```
$ git config --list
```

Still have to introduce yourself? To that end, we set our user name and email address like this:

```
$ git config --global user.name 'simonmunzert'
$ git config --global user.email 'simon.munzert@example.com'
```

The user name can be (but does not have to be) your GitHub user name. The email address should definitely be the one associated with your GitHub account.

Check out these setup instructions from Software Carpentry to learn about more configuration options.

Git(Hub) + RStudio

Link a GitHub repo to an RStudio Project

One of the (many) great features of RStudio is how well it integrates version control into your everyday workflow.

- Even though Git is a completely separate program to R, they feel like part of the same "thing" in RStudio.
- This next section is about learning the basic Git(Hub) commands and the recipe for successful project integration with RStudio.

Link a GitHub repo to an RStudio Project

One of the (many) great features of RStudio is how well it integrates version control into your everyday workflow.

- Even though Git is a completely separate program to R, they feel like part of the same "thing" in RStudio.
- This next section is about learning the basic Git(Hub) commands and the recipe for successful project integration with RStudio.

The starting point for our workflow is to link a GitHub repository (i.e. "repo") to an RStudio Project. Here are the steps we're going to follow:

1. Create the repo on GitHub and initialize with a README.
2. Copy the HTTPS/SSH link (the green "Clone or Download" button).
3. Open up RStudio.
4. Navigate to **File -> New Project -> Version Control -> Git**.
5. Paste your copied link into the "Repository URL:" box.
6. Choose the project path ("Create project as subdirectory of:") and click **Create Project**.

Link a GitHub repo to an RStudio Project

One of the (many) great features of RStudio is how well it integrates version control into your everyday workflow.

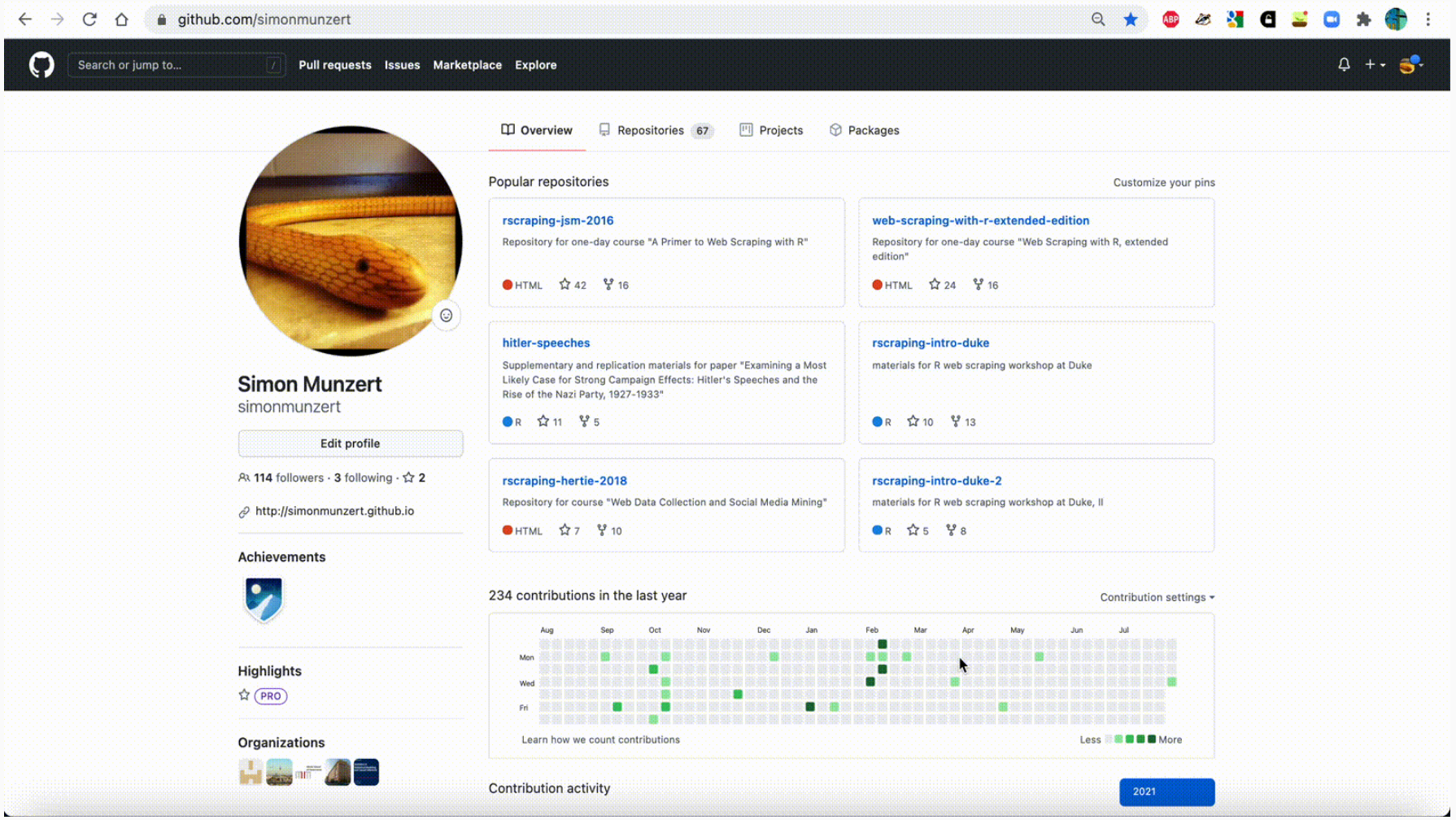
- Even though Git is a completely separate program to R, they feel like part of the same "thing" in RStudio.
- This next section is about learning the basic Git(Hub) commands and the recipe for successful project integration with RStudio.

The starting point for our workflow is to link a GitHub repository (i.e. "repo") to an RStudio Project. Here are the steps we're going to follow:

1. Create the repo on GitHub and initialize with a README.
2. Copy the HTTPS/SSH link (the green "Clone or Download" button).
3. Open up RStudio.
4. Navigate to **File -> New Project -> Version Control -> Git**.
5. Paste your copied link into the "Repository URL:" box.
6. Choose the project path ("Create project as subdirectory of:") and click **Create Project**.

Let's see how that works in practice.

Link a GitHub repo to an RStudio Project



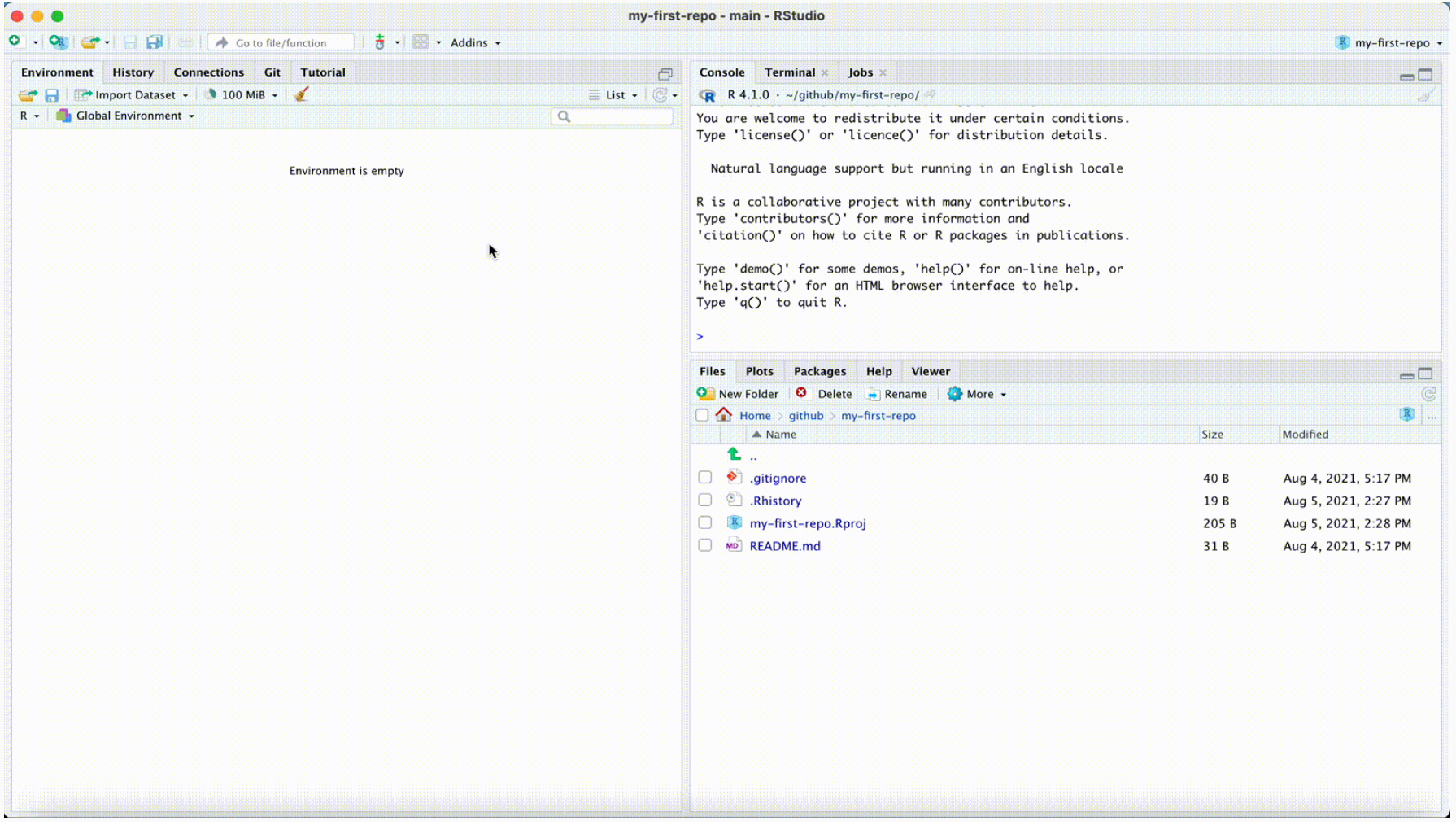
Make some local changes

Look at the Environment panel in your RStudio IDE. Do you see the "Git" tab?

1. Click on it.
2. Look what's in there. There should already be some files in there, which we'll ignore for the moment.
3. Now open up the README file (see the "Files" tab in the bottom-right panel).
4. Add some text like "Hello World!" and save the README.
5. Have you noticed the changes in the "Git" panel?

Again, see the GIF walkthrough on the next slide...

Make some local changes



Main Git operations

Now that you've cloned your first repo and made some local changes, it's time to learn the four main Git operations.

1. **Stage** (or "add")
   - Tell Git that you want to add changes to the repo history (file edits, additions, deletions, etc.)
2. **Commit**
   - Tell Git that, yes, you are sure these changes should be part of the repo history.
3. **Push**
   - Push any (committed) local changes to the GitHub repo.
4. **Pull**
   - Get any new changes made on the GitHub repo (i.e. the upstream remote), either by your collaborators or you on another machine.

Main Git operations

Now that you've cloned your first repo and made some local changes, it's time to learn the four main Git operations.

1. **Stage** (or "add")
   - Tell Git that you want to add changes to the repo history (file edits, additions, deletions, etc.)
2. **Commit**
   - Tell Git that, yes, you are sure these changes should be part of the repo history.
3. **Push**
   - Push any (committed) local changes to the GitHub repo.
4. **Pull**
   - Get any new changes made on the GitHub repo (i.e. the upstream remote), either by your collaborators or you on another machine.

For the moment, it will be useful to group the first two operations and last two operations together. (They are often combined in practice too, although you'll soon get a sense of when and why they should be split up.)

Main Git operations

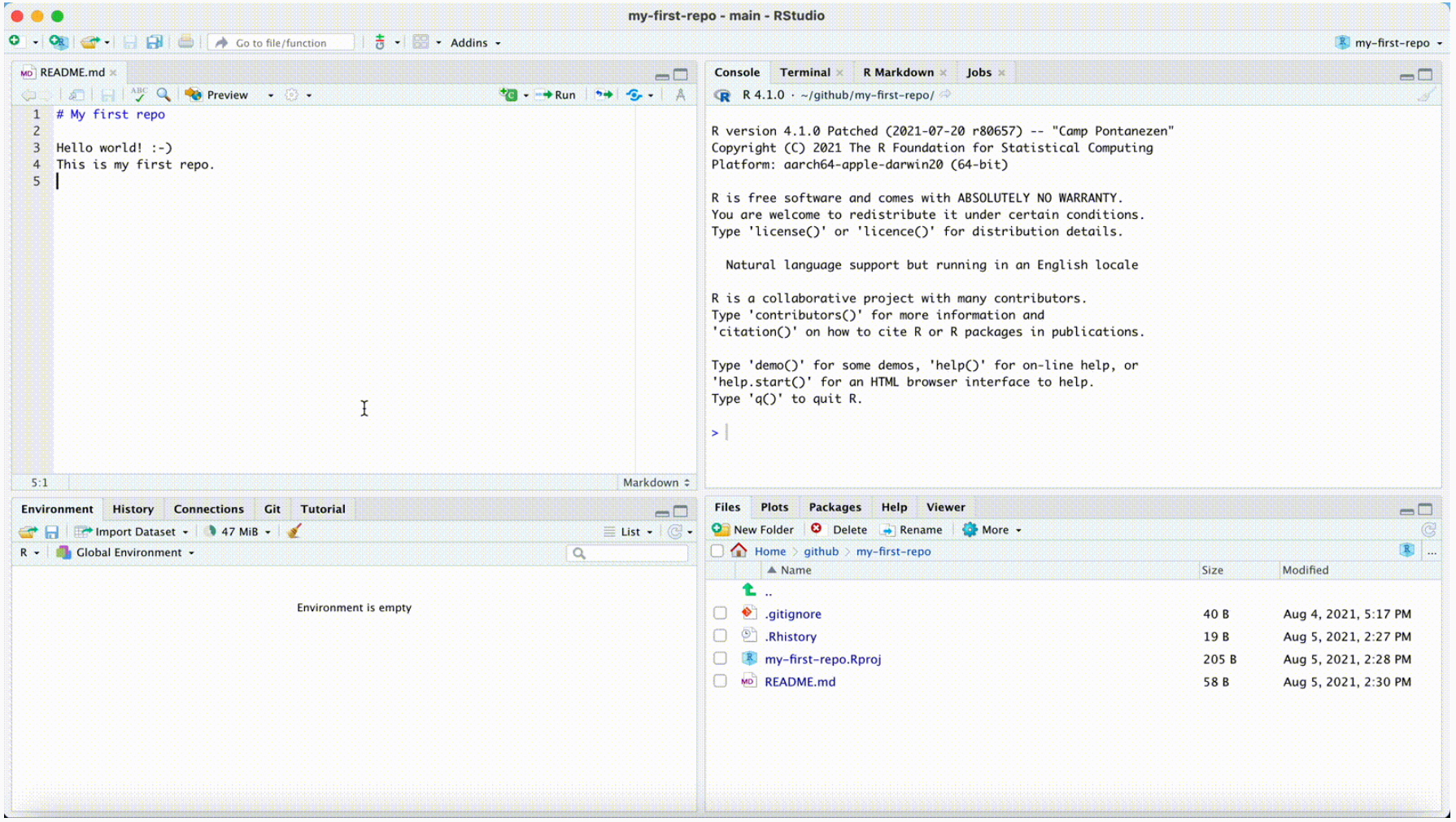
Now that you've cloned your first repo and made some local changes, it's time to learn the four main Git operations.

1. **Stage** (or "add")
    - Tell Git that you want to add changes to the repo history (file edits, additions, deletions, etc.)
2. **Commit**
    - Tell Git that, yes, you are sure these changes should be part of the repo history.
3. **Push**
    - Push any (committed) local changes to the GitHub repo.
4. **Pull**
    - Get any new changes made on the GitHub repo (i.e. the upstream remote), either by your collaborators or you on another machine.

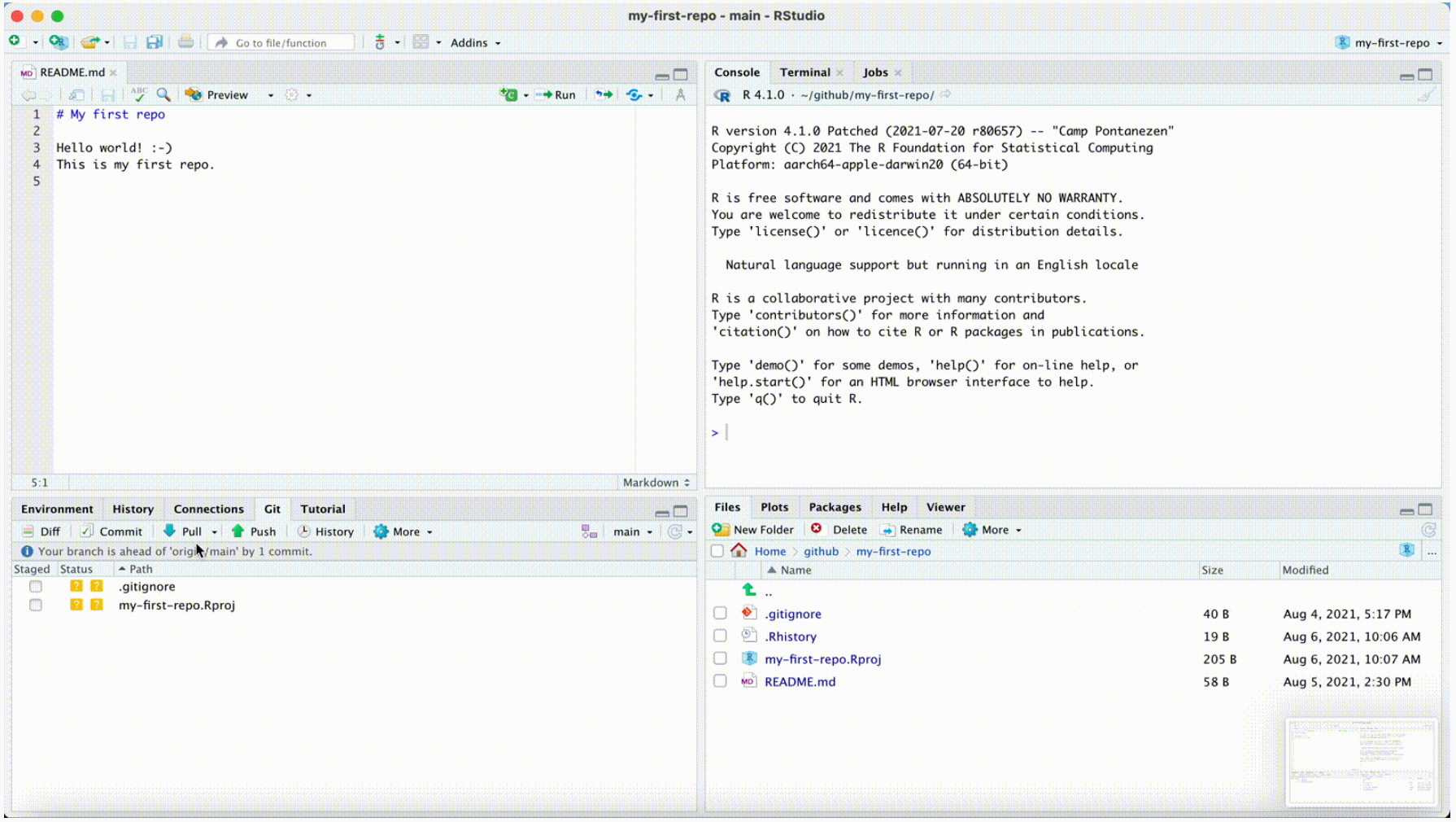
For the moment, it will be useful to group the first two operations and last two operations together. (They are often combined in practice too, although you'll soon get a sense of when and why they should be split up.)

Ready for more GIFs?

Stage and Commit



Push and Pull



Recap

Here's a step-by-step summary of what we just did.

- Made some changes to a file and saved them locally
- *Added*/*Staged* these local changes.
- *Committed* these local changes to our Git history with a helpful message.
- *Pulled* from the GitHub repo just in case anyone else made changes too (not expected here, but good practice).
- *Pushed* our changes to the GitHub repo.

Recap

Here's a step-by-step summary of what we just did.

- Made some changes to a file and saved them locally
- *Added*/*Staged* these local changes.
- *Committed* these local changes to our Git history with a helpful message.
- *Pulled* from the GitHub repo just in case anyone else made changes too (not expected here, but good practice).
- *Pushed* our changes to the GitHub repo.

🦊 Always pull from the upstream repo *before* you push any changes. Seriously, do this even on solo projects; making it a habit will save you headaches down the road.

Recap

Here's a step-by-step summary of what we just did.

- Made some changes to a file and saved them locally
- *Added*/*Staged* these local changes.
- *Committed* these local changes to our Git history with a helpful message.
- *Pulled* from the GitHub repo just in case anyone else made changes too (not expected here, but good practice).
- *Pushed* our changes to the GitHub repo.

🦊 Always pull from the upstream repo *before* you push any changes. Seriously, do this even on solo projects; making it a habit will save you headaches down the road.

🤓 You were likely challenged for your GitHub credentials at some point. Learn how to cache these here. Note that the classical username/password authentication was recently removed. Instead, you have to use your personal access token (PAT). I recommend `usethis::create_github_token()` in combination with `gitcreds::gitcreds_set()`.

Recap

Here's a step-by-step summary of what we just did.

- Made some changes to a file and saved them locally
- *Added*/*Staged* these local changes.
- *Committed* these local changes to our Git history with a helpful message.
- *Pulled* from the GitHub repo just in case anyone else made changes too (not expected here, but good practice).
- *Pushed* our changes to the GitHub repo.

🦊 Always pull from the upstream repo *before* you push any changes. Seriously, do this even on solo projects; making it a habit will save you headaches down the road.

🤓 You were likely challenged for your GitHub credentials at some point. Learn how to cache these here. Note that the classical username/password authentication was recently removed. Instead, you have to use your personal access token (PAT). I recommend `usethis::create_github_token()` in combination with `gitcreds::gitcreds_set()`.

🤓 Speaking of credentials, another approach is to switch to SSH. You might want to consider doing this when you feel comfortable with the main Git operations.

Why this workflow?

Creating the repo on GitHub first means that it will always be "upstream" of your (and any other) local copies.

- In effect, this allows GitHub to act as the central node in the distributed VC network.
- Especially valuable when you are collaborating on a project with others — more on this later — but also has advantages when you are working alone.
- If you would like to move an existing project to GitHub, my advice is still to create an empty repo there first, clone it locally, and then copy all your files across.

Why this workflow?

Creating the repo on GitHub first means that it will always be "upstream" of your (and any other) local copies.

- In effect, this allows GitHub to act as the central node in the distributed VC network.
- Especially valuable when you are collaborating on a project with others — more on this later — but also has advantages when you are working alone.
- If you would like to move an existing project to GitHub, my advice is still to create an empty repo there first, clone it locally, and then copy all your files across.

RStudio Projects are great.

- Again, they interact seamlessly with Git(Hub), as we've just seen.
- They also solve absolute vs. relative path problems, since the `.Rproj` file acts as an anchor point for all other files in the repo.[1]

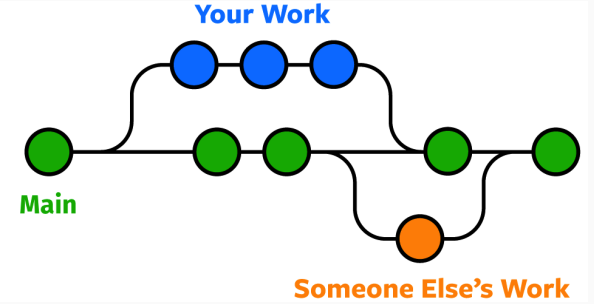
[1] You know that calling files from `YourComputer/YourName/Documents/Special-Subfolder/etc` in your scripts is one of the deadly sins of programming, right?

Branches and forking

What are branches and why use them?

Branches are cool.

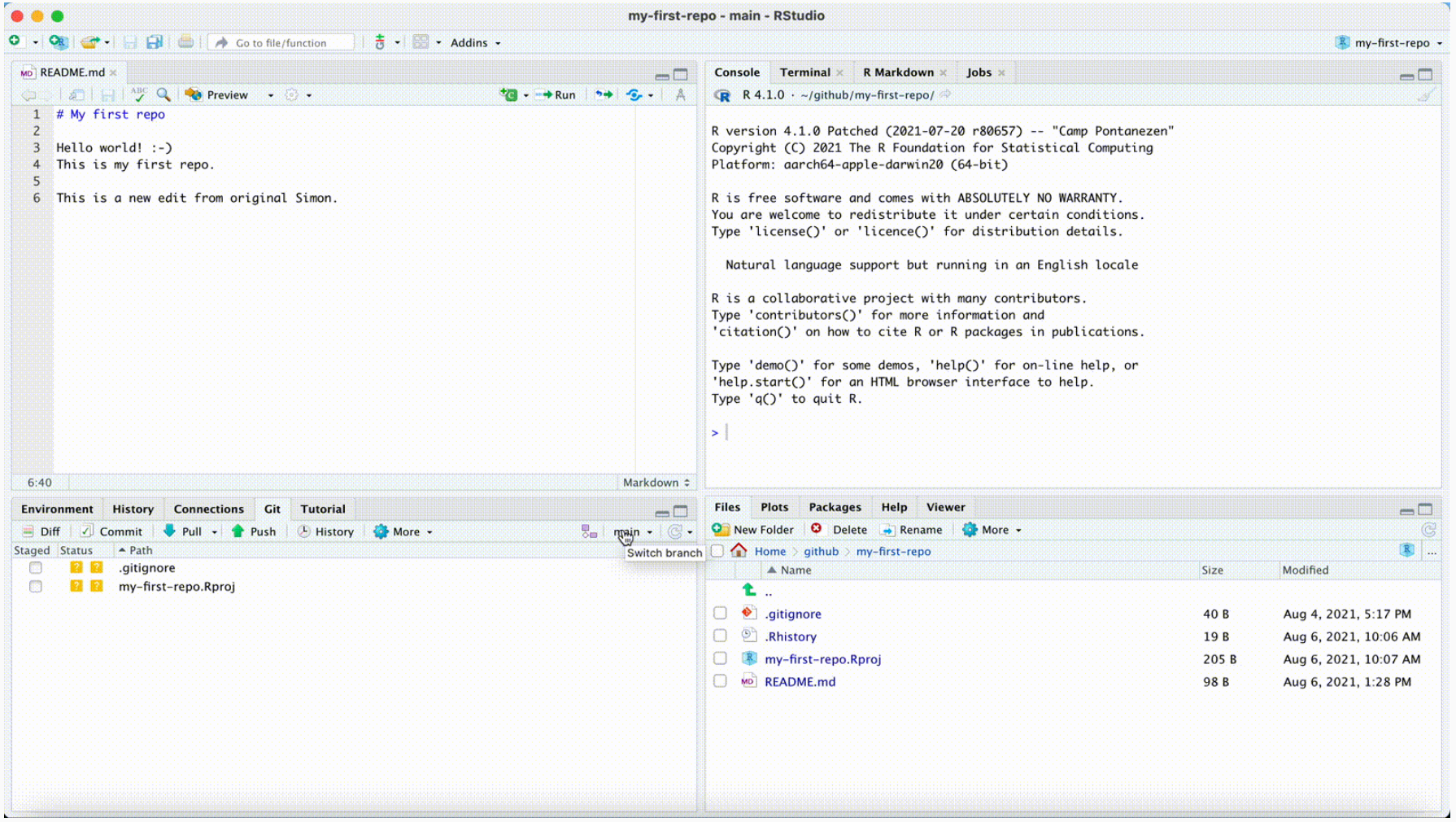
- Think of your repo as a growing tree.
- Branches allow you to take a snapshot of your existing repo and try out a whole new idea *without affecting* your main (formerly "master") branch.
- Only once you (and your collaborators) are 100% satisfied, would you merge it back into the master branch.[1]
  - This is how most new features in modern software and apps are developed.
  - It is also how bugs are caught and fixed.
  - But researchers can easily use it to try out new ideas and analysis (e.g. robustness checks, revisions, etc.).
- If you aren't happy, then you can just delete the experimental branch and continue as if nothing happened.



`Credit` noble desktop

[1] As with real trees, you can actually have branches of branches (of branches). But let's not get ahead of ourselves.

Create a new branch in RStudio



Merging branches + Pull requests

You have two options:

Merging branches + Pull requests

You have two options:

1. Locally

- Commit your final changes to the new branch (say we call it "new-idea").
- Switch back to the master branch: `$ git checkout master`
- Merge in the new-idea branch changes: `$ git merge new-idea`
- Delete the new-idea branch (optional): `$ git branch -d new-idea`

Merging branches + Pull requests

You have two options:

1. Locally

- Commit your final changes to the new branch (say we call it "new-idea").
- Switch back to the master branch: `$ git checkout master`
- Merge in the new-idea branch changes: `$ git merge new-idea`
- Delete the new-idea branch (optional): `$ git branch -d new-idea`

2. Remotely (i.e. *pull requests* on GitHub)

- PRs are a way to notify collaborators — or yourself! — that you have completed a feature.
- You write a summary of all the changes contained in the branch.
- You then assign suggested reviewers of your code — including yourself potentially — who are then able to approve these changes ("Merge pull request") on GitHub.

Merging branches + Pull requests

You have two options:

1. Locally

- Commit your final changes to the new branch (say we call it "new-idea").
- Switch back to the master branch: `$ git checkout master`
- Merge in the new-idea branch changes: `$ git merge new-idea`
- Delete the new-idea branch (optional): `$ git branch -d new-idea`

2. Remotely (i.e. *pull requests* on GitHub)

- PRs are a way to notify collaborators — or yourself! — that you have completed a feature.
- You write a summary of all the changes contained in the branch.
- You then assign suggested reviewers of your code — including yourself potentially — who are then able to approve these changes ("Merge pull request") on GitHub.

Let's see how that works in practice.

Your first pull request

Enter new branch

- First, operate in the new branch.
- Switch over to it if you haven't already.
- Remember: `$ git checkout new-idea` (or just click on the branches tab in RStudio)

Your first pull request

Enter new branch

- First, operate in the new branch.
- Switch over to it if you haven't already.
- Remember: `$ git checkout new-idea` (or just click on the branches tab in RStudio)

Make edits

- Make some local changes and then commit + push them to GitHub.
- For a start, the changes themselves don't really matter. Add text to the README, add some new files, whatever.

Your first pull request

Enter new branch

- First, operate in the new branch.
- Switch over to it if you haven't already.
- Remember: `$ git checkout new-idea` (or just click on the branches tab in RStudio)

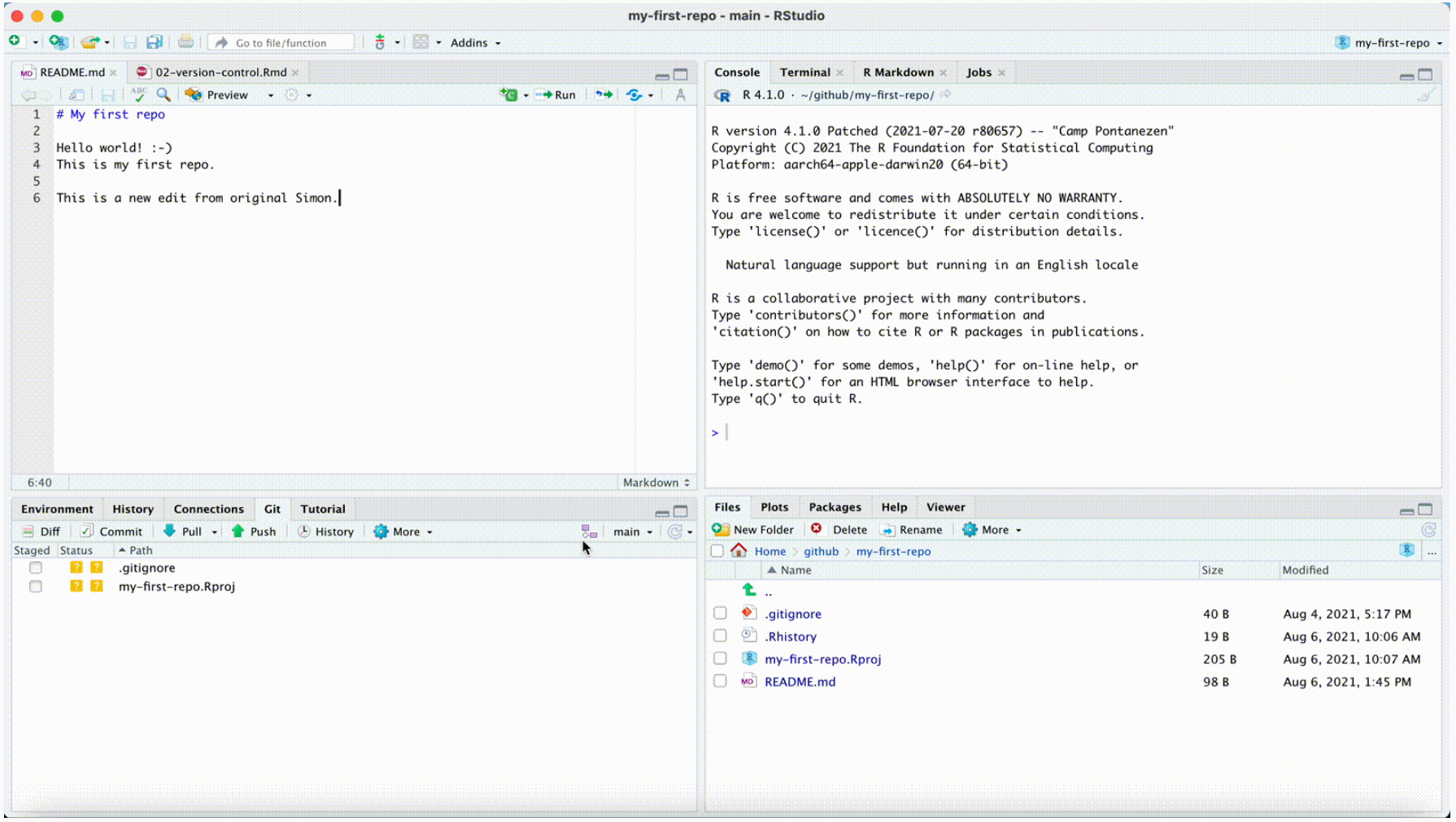
Make edits

- Make some local changes and then commit + push them to GitHub.
- For a start, the changes themselves don't really matter. Add text to the README, add some new files, whatever.

Create pull request

- After pushing these changes, head over to your repo on GitHub.
- You should see a new green button with "Compare & pull request". Click it.
- Add a meta description of what this PR accomplishes. You can also change the title if you want.
- Click "Create pull request".
- (Here's where you or your collaborators would review all the changes.)
- Once satisfied, click "Merge pull request" and then confirm.

Your first pull request (cont.)



Forks

What are forks?

- Git forks lie somewhere between cloning a repo and branching from it.
- In fact, if you fork a repo then you are really creating a copy of it.

Forks

What are forks?

- Git forks lie somewhere between cloning a repo and branching from it.
- In fact, if you fork a repo then you are really creating a copy of it.

How does it work?

- Forking a repo on GitHub is very simple; just click the "Fork" button in the top-right corner of said repo.
- This will create an independent copy of the repo under your GitHub account.

Forks

What are forks?

- Git forks lie somewhere between cloning a repo and branching from it.
- In fact, if you fork a repo then you are really creating a copy of it.

How does it work?

- Forking a repo on GitHub is very simple; just click the "Fork" button in the top-right corner of said repo.
- This will create an independent copy of the repo under your GitHub account.

What to do with a forked repo?

- Once you fork a repo, you are free to do anything you want to it. (It's yours.) However, forking — in combination with pull requests — is actually how much of the world's software is developed. For example:
  - Outside user *B* forks *A*'s repo. She adds a new feature (or fixes a bug she's identified) and then issues an upstream pull request.
  - *A* is notified and can then decide whether to merge *B*'s contribution with the main project.
- If the original repo is still under active development, you might want to stay up to date. Learn more about syncing a fork here.

Merge conflicts

When things don't match up

The context

- While version control also makes sense when you work alone, it becomes essential when you collaborate with others.
- When multiple people work on the same code simultaneously on different machines, this can lead to problems.
- One way to avoid these problems is to simply not work on code simultaneously. This is an option in small teams, but not in bigger projects.

When things don't match up

The context

- While version control also makes sense when you work alone, it becomes essential when you collaborate with others.
- When multiple people work on the same code simultaneously on different machines, this can lead to problems.
- One way to avoid these problems is to simply not work on code simultaneously. This is an option in small teams, but not in bigger projects.

The problem

- Merge conflicts occur when Git cannot resolve differences in the code between two commits.
- Not all code differences result in merge conflicts. When changes in code occur on different lines or in different files, Git will successfully merge commits.
- The fact that merge conflicts can occur is not a bug, but a key feature of Git! It gives you the agency to address them properly.

A hypothetical example

Think of two collaborators, C1 and C2. You are C1.

- C1: You invite C2 to join you as a collaborator on the "test" GitHub repo that you created earlier. (See the *Settings* tab of your repo, then → *Manage access*.)
- C2: Clones C1's repo to their local machine. Makes some edits to the README (e.g., deletes lines of text and adds their own). Stages, commits and pushes these changes.
- C1: You make your own changes to the README on your local machine. Stage, commit and then try to push them (*after* pulling from the GitHub repo first).
- C1 encounters a `merge conflict` error that needs to be fixed.

A hypothetical example

Think of two collaborators, C1 and C2. You are C1.

- C1: You invite C2 to join you as a collaborator on the "test" GitHub repo that you created earlier. (See the *Settings* tab of your repo, then → *Manage access*.)
- C2: Clones C1's repo to their local machine. Makes some edits to the README (e.g., deletes lines of text and adds their own). Stages, commits and pushes these changes.
- C1: You make your own changes to the README on your local machine. Stage, commit and then try to push them (*after* pulling from the GitHub repo first).
- C1 encounters a `merge conflict` error that needs to be fixed.

🤓 Note what Git is doing here: It protects your contribution by refusing to merge. It wants to make sure that you don't accidentally overwrite all of your changes by pulling C2's version of the modified file.

A hypothetical example

Think of two collaborators, C1 and C2. You are C1.

- C1: You invite C2 to join you as a collaborator on the "test" GitHub repo that you created earlier. (See the *Settings* tab of your repo, then → *Manage access*.)
- C2: Clones C1's repo to their local machine. Makes some edits to the README (e.g., deletes lines of text and adds their own). Stages, commits and pushes these changes.
- C1: You make your own changes to the README on your local machine. Stage, commit and then try to push them (*after* pulling from the GitHub repo first).
- C1 encounters a `merge conflict` error that needs to be fixed.

🤓 Note what Git is doing here: It protects your contribution by refusing to merge. It wants to make sure that you don't accidentally overwrite all of your changes by pulling C2's version of the modified file.

🦊 In toy examples like these, the source of the problem is obvious. In bigger projects, `git status` can provide a helpful summary to see which files are in conflict.

Interpreting merge conflicts

Let's see what's happening here by opening up the README file in RStudio/your preferred text editor. You should see something like:

```
# README
Some text here.
<<<<<<< HEAD
Text added by Collaborator 2.
=======
Text added by Collaborator 1.
>>>>>>> 814e09178910383c128045ce67a58c9c1df3f558.
More text here.
```

Interpreting merge conflicts

Let's see what's happening here by opening up the README file in RStudio/your preferred text editor. You should see something like:

```
# README
Some text here.
<<<<<<< HEAD
Text added by Collaborator 2.
=======
Text added by Collaborator 1.
>>>>>>> 814e09178910383c128045ce67a58c9c1df3f558.
More text here.
```

What do these symbols mean?

Interpreting merge conflicts

Let's see what's happening here by opening up the README file in RStudio/your preferred text editor. You should see something like:

```
# README
Some text here.
<<<<<<< HEAD
Text added by Collaborator 2.
=======
Text added by Collaborator 1.
>>>>>>> 814e09178910383c128045ce67a58c9c1df3f558.
More text here.
```

What do these symbols mean?

- `<<<<<<< HEAD` Indicates the start of the merge conflict.

Interpreting merge conflicts

Let's see what's happening here by opening up the README file in RStudio/your preferred text editor. You should see something like:

```
# README
Some text here.
<<<<<<< HEAD
Text added by Collaborator 2.
=======
Text added by Collaborator 1.
>>>>>>> 814e09178910383c128045ce67a58c9c1df3f558.
More text here.
```

What do these symbols mean?

- `<<<<<<< HEAD` Indicates the start of the merge conflict.
- `=======` Indicates the break point used for comparison.

Interpreting merge conflicts

Let's see what's happening here by opening up the README file in RStudio/your preferred text editor. You should see something like:

```
# README
Some text here.
<<<<<<< HEAD
Text added by Collaborator 2.
=======
Text added by Collaborator 1.
>>>>>>> 814e09178910383c128045ce67a58c9c1df3f558.
More text here.
```

What do these symbols mean?

- `<<<<<<< HEAD` Indicates the start of the merge conflict.
- `=======` Indicates the break point used for comparison.
- `>>>>>>> <long string>` Indicates the end of the lines that had a merge conflict.

Fixing merge conflicts

How merge conflicts are fixed

- Fixing these conflicts is a matter of (manually) editing the README file.
- Delete the lines of the text that you don't want.
- Then, delete the special Git merge conflict symbols.
- Once that's done, you should be able to stage, commit, pull and finally push your changes to the GitHub repo without any errors.

Fixing merge conflicts

How merge conflicts are fixed

- Fixing these conflicts is a matter of (manually) editing the README file.
- Delete the lines of the text that you don't want.
- Then, delete the special Git merge conflict symbols.
- Once that's done, you should be able to stage, commit, pull and finally push your changes to the GitHub repo without any errors.

Caveats

- C1 gets to decide what to keep because they fixed the merge conflict.
- But: The full commit history is preserved, so C2 can always recover their changes if desired.
- A more elegant and democratic solution to merge conflicts (and repo changes in general) is provided by Git **branches**.

Aside: Line endings and different OSs

Problem

During your collaboration, you may have encountered a situation where Git is highlighting differences on seemingly unchanged sentences.

- If that is the case, check whether your partner is using a different OS to you.

The "culprit" is the fact that Git evaluates an invisible character at the end of every line. This is how Git tracks changes.

- For Linux and MacOS, that ending is "LF".
- For Windows, that ending is "CRLF".
- Check out this Wikipedia article for valuable cocktail party knowledge about where these terms come from.

Aside: Line endings and different OSs

Problem

During your collaboration, you may have encountered a situation where Git is highlighting differences on seemingly unchanged sentences.

- If that is the case, check whether your partner is using a different OS to you.

The "culprit" is the fact that Git evaluates an invisible character at the end of every line. This is how Git tracks changes.

- For Linux and MacOS, that ending is "LF".
- For Windows, that ending is "CRLF".
- Check out this Wikipedia article for valuable cocktail party knowledge about where these terms come from.

Solution

Open up the shell and enter

```
$ git config --global core.autocrlf input
```

(Windows users: Change `input` to `true`).

Git from the shell

Why bother with the shell?

The GitHub + RStudio Project combo is ideal for new users.

- RStudio's Git integration and built-in GUI cover all the major operations.
- RStudio Projects FTW.

However, there are **some benefits of the shell**:

- The shell is more powerful and flexible. It lets you do things that the RStudio Git GUI can't.
- Working in the shell is potentially more appropriate for projects that aren't primarily based in R.
- Knowing the basic Git commands in the shell is generally a good thing for a data scientist.
- For a hands-on intro to the shell/command line, check out the optional session in our GitHub organization.

Git clone/status

Clone a repo into current directory.

```
$ git clone REPOSITORY-URL
```

See the commit history (hit spacebar to scroll down or q to exit). You need to be in a git repo to see something.

```
$ git log
```

What has changed?

```
$ git status
```

Git stage

Stage ("add") a file or group of files.

```
$ git add NAME-OF-FILE-OR-FOLDER
```

You can use wildcard characters to stage a group of files (e.g. sharing a common prefix). There are a bunch of useful flag options too:

- Stage all files.

  ```
  $ git add -A
  ```

- Stage updated files only (modified or deleted, but not new).

  ```
  $ git add -u
  ```

- Stage new files only (not updated).

  ```
  $ git add .
  ```

Git commit/pull/push

Commit your changes.

```
$ git commit -m "Helpful message"
```

Pull from the upstream repository (i.e. GitHub).

```
$ git pull
```

Push any local changes that you've committed to the upstream repo (i.e. GitHub).

```
$ git push
```

Git checkout/branch

Create a new branch on your local machine and switch to it:

```
$ git checkout -b NAME-OF-YOUR-NEW-BRANCH
```

Push the new branch to GitHub:

```
$ git push origin NAME-OF-YOUR-NEW-BRANCH
```

List all branches on your local machine:

```
$ git branch
```

Switch back to (e.g.) the main branch:

```
$ git checkout main
```

Delete a branch

```
$ git branch -d NAME-OF-YOUR-FAILED-BRANCH
$ git push origin :NAME-OF-YOUR-FAILED-BRANCH
```

Other tips

README

Why README?

- README files are special in GitHub because they act as repo landing pages.
- They're the first thing viewers of your repo see and should therefore communicate what the repo is about.

README

Why README?

- README files are special in GitHub because they act as repo landing pages.
- They're the first thing viewers of your repo see and should therefore communicate what the repo is about.

Where README?

- For a project tied to a research paper, this is where you should be explicit about the goal of the research paper, the software requirements, how to run the analysis, and so forth (e.g. here).
- On the other end of the scale, many GitHub repos are basically standalone README files. Think of these as version-controlled blog posts (e.g. here).
- R packages that are published on GitHub often provide more useful information in the README (and in the repo in general) than on CRAN (if they're published there at all). See, e.g., here vs. here.

README

Why README?

- README files are special in GitHub because they act as repo landing pages.
- They're the first thing viewers of your repo see and should therefore communicate what the repo is about.

Where README?

- For a project tied to a research paper, this is where you should be explicit about the goal of the research paper, the software requirements, how to run the analysis, and so forth (e.g. here).
- On the other end of the scale, many GitHub repos are basically standalone README files. Think of these as version-controlled blog posts (e.g. here).
- R packages that are published on GitHub often provide more useful information in the README (and in the repo in general) than on CRAN (if they're published there at all). See, e.g., here vs. here.

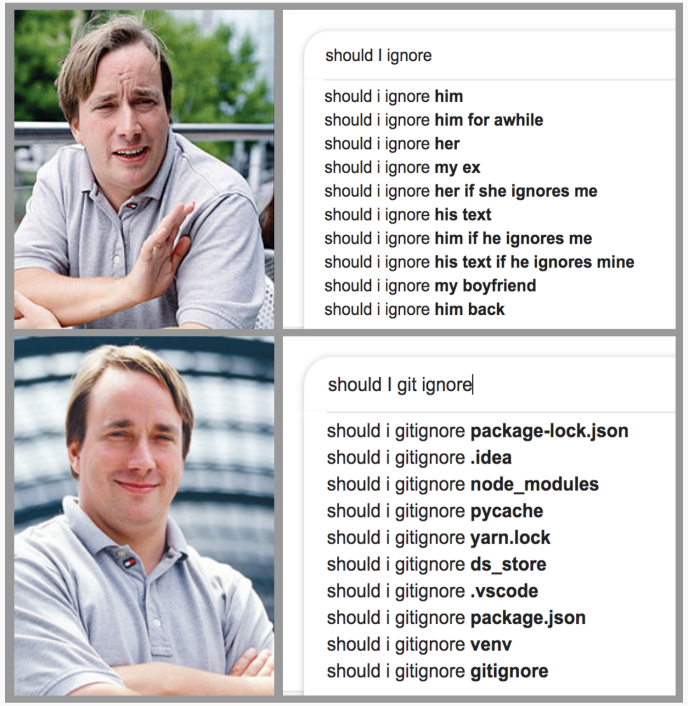
How README?

- READMEs should be written in Markdown, which GH automatically renders.
- You'll learn more about Markdown (and its cousin, R Markdown) during the course of our homework assignments.

.gitignore

What is .gitignore?

- A `.gitignore` file tells Git what to ignore.
- This is especially useful if you want to exclude whole folders or a class of files (e.g. based on size or type).
- Proprietary data files should be ignored from the beginning if you intend to make a repo public at some point.
- Very large individual files (>100 MB) exceed GitHub's maximum allowable size and should be ignored regardless. See here and here.



`Credit` RulerD/reddit.com

.gitignore (cont.)

Multiple ways to create a .gitignore

- A `.gitignore` file was automatically generated if you cloned your repo with an RStudio Project.
- You could also have the option of adding one when you first create a repo on GitHub.
- Or, you can create one with your preferred text editor. (Must be saved as "`.gitignore`".)

.gitignore (cont.)

Multiple ways to create a .gitignore

- A `.gitignore` file was automatically generated if you cloned your repo with an RStudio Project.
- You could also have the option of adding one when you first create a repo on GitHub.
- Or, you can create one with your preferred text editor. (Must be saved as "`.gitignore`".)

How to specify rules in the .gitignore file

- To ignore a single a file: `FILE-I-WANT-TO-IGNORE.csv`
- To ignore a whole folder (and all of its contents, subfolders, etc.): `FOLDER-NAME/**`
- The standard shell commands and special characters apply.
  - E.g., ignore all CSV files in the repo: `*.csv`
  - E.g., ignore all files beginning with "test": `test*`
  - E.g., don't ignore a particular file: `!somefile.txt`

GitHub Issues

What is a GitHub Issue?

- GitHub Issues are another great way to interact with your collaborators and/or package maintainers.
- They are a genuine feature of GitHub (not Git).
- Think of it as a communication board where you can flag bugs, request new features, suggest changes, etc.
- Repo maintainers can keep the issues as a to do list (and close issues that have been "solved"), but also communicate with people who brought up the issue.

GitHub Issues

What is a GitHub Issue?

- GitHub Issues are another great way to interact with your collaborators and/or package maintainers.
- They are a genuine feature of GitHub (not Git).
- Think of it as a communication board where you can flag bugs, request new features, suggest changes, etc.
- Repo maintainers can keep the issues as a to do list (and close issues that have been "solved"), but also communicate with people who brought up the issue.

How to deal with issues?

- Before filing an issue, you should check the list of open (and maybe also closed) issues.
- If you spot any problems with these lecture notes, feel free to file an issue here!

Summary

Recipe (shell commands in grey)

1. Create a repo on GitHub and initialize with a README.
2. Clone the repo to your local machine. Preferably using an RStudio Project, but as you wish. (E.g. Shell command: `$ git clone REPOSITORY-URL`)
3. Stage any changes you make: `$ git add -A`
4. Commit your changes: `$ git commit -m "Helpful message"`
5. Pull from GitHub: `$ git pull`
6. (Fix any merge conflicts.)
7. Push your changes to GitHub: `$ git push`

Recipe (shell commands in grey)

1. Create a repo on GitHub and initialize with a README.
2. Clone the repo to your local machine. Preferably using an RStudio Project, but as you wish. (E.g. Shell command: `$ git clone REPOSITORY-URL`)
3. Stage any changes you make: `$ git add -A`
4. Commit your changes: `$ git commit -m "Helpful message"`
5. Pull from GitHub: `$ git pull`
6. (Fix any merge conflicts.)
7. Push your changes to GitHub: `$ git push`

Repeat steps 3—7 (but especially steps 3 and 4) often.

FAQ

Q: When should I use version control with Git? Is DropBox/OneDrive/Google Docs not enough?

A: It's not evil to use these tools.

- Depending on the size and complexity of a project, version history of cloud storage services might seem sufficient (and more convenient to use!).
- But as an aspiring data scientist, Git/GitHub is going to become part of your workflow. And it will be less painful the more experience you have.

Q: Should I try to work in the shell, or are convenient GUIs as given by RStudio, GitHub Desktop etc. fine, too?

A: Again, it's not evil to use these tools.

- Start simple and stay within the RStudio IDE.
- Return to the shell once you've learned more.

Q: When should I commit (and push) changes?

A: Early and often.

- It's not quite as important as saving your work regularly, but it's a close second.
- You should certainly push everything that you want your collaborators to see.

Q: Do I need branches if I am working on a solo project?

A: You don't *need* them, but they offer big advantages in maintaining a sane workflow.

- Experiment without any risk to the main project!
- If you combine them with pull requests, then you can compress significant additions to your project (which may comprise many small edits) into a single branch.

FAQ (cont.)

Q: What's the difference between cloning and forking a repo?

A: Cloning directly ties your local version to the original repo, while forking creates a copy on your GitHub (which you can then clone).

- Cloning makes it easier to fetch updates (and is often the best choice for new GitHub users), but forking has advantages too.

Q: What happens when something goes wrong?

A: Think: "Oh shit, Git!" and check out http://ohshitgit.com/.

Q: What happens when something goes *horribly* wrong?

A: Burn it down and start again.

- http://happygitwithr.com/burn.html
- This is a great advantage of Git's distributed nature. If something goes horribly wrong, there's usually an intact version somewhere else.

Coming up

Get the course materials

If you haven't done so already, now is a good time to clone/fork the course materials to your local computer.

The first assignment

Now that you've learned the necessary basics, Mock Assignment 0 is up on GitHub Classroom. Check it out to learn how to work on and submit assignments!

Next lecture

R and the tidyverse, and: good coding style!