Data Science for Economists

Lecture 2: Version control with Git(Hub)

Grant McDermott, adapted by Kyle Coombs
Bates College | ECON 368

Table of contents

1. Prologue
2. Git and GitHub
3. Git(Hub) + RStudio
4. Git from the shell
5. Merge conflicts
6. Branches and forking
7. Other tips
8. Summary
9. Appendix: FAQ

Prologue

Goal today

1. Introduce version control

2. Learn the steps of a basic Git/GitHub workflow with Rstudio

    - Create a PAT in RStudio if you did not complete the first exercise.
    - Create a GitHub repository as a project in RStudio.
    - Implement a commit, push, and pull using GitHub.
    - Create a new branch in your repository.
    - Create and fix a merge conflict in your repository.

3. Talk about READMEs and .gitignore files

Before we start

We went through a software installation check during the previous lecture. By now you should have:

☑ Installed R.

☑ Installed RStudio.

☑ Installed Git and GitHub Desktop.

☑ Created an account on GitHub

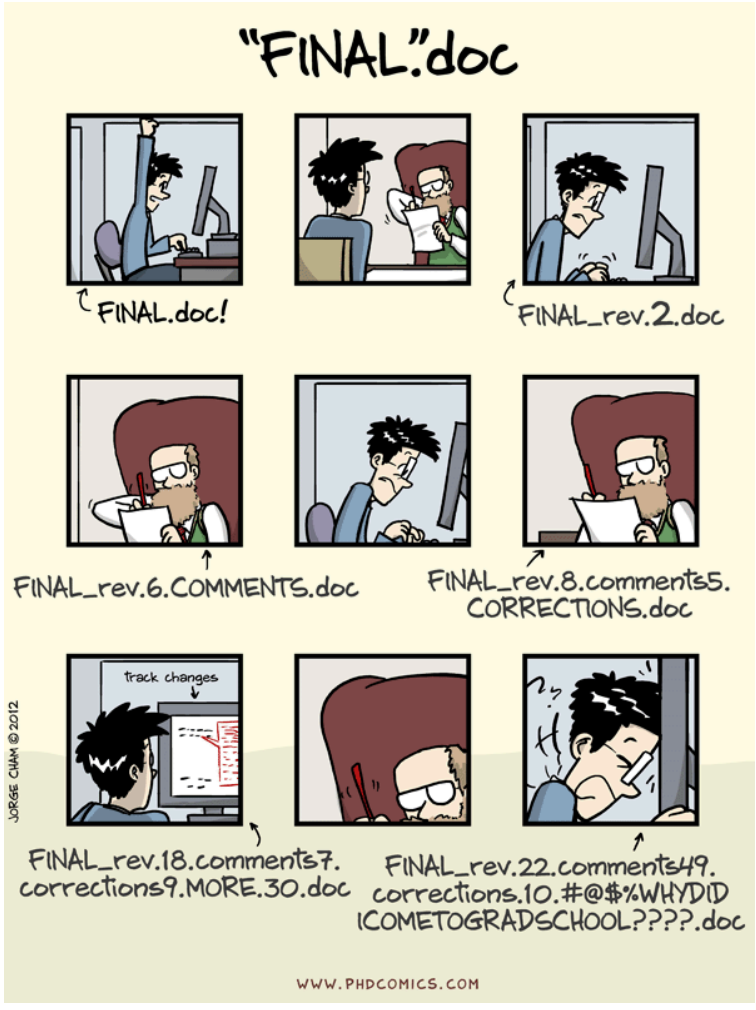
☑ Accepted an invitation to ECON 368 course repo.

☑ Created a PAT in RStudio and added it to your GitHub account.

If in doubt about software, please consult Jenny Bryan's amazing guide: http://happygitwithr.com.

Git and GitHub

Why bother?


"FINAL".doc
FINAL.doc!
FINAL_rev.2.doc
FINAL_rev.6.COMMENTS.doc
FINAL_rev.8.comments5.
CORRECTIONS.doc
track changes
FINAL_rev.18.comments7.
corrections9.MORE.30.doc
FINAL_rev.22.comments49.
corrections.10.#@$%WHYDID
ICOMETOGRADSCHOOL????.doc
JORGE CHAM © 2012
WWW.PHDCOMICS.COM

Git(Hub) solves this problem

Git

- Git is a distributed version control system. (Wait, what?)
- Okay, try this: Imagine if Dropbox and the "Track changes" feature in MS Word had a baby. Git would be that baby.
- Git is optimized for coding and project management, i.e. what economists and data scientists spend a lot of time doing
- There is a learning curve, but I promise you it's worth it.

GitHub

- Git and GitHub are distinct things.
- GitHub is an online hosting platform that provides an array of services built on top of the Git system. (Similar platforms include Bitbucket and GitLab.)
- We don't *need* GitHub to use Git... But it will make our lives so much easier.

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

- Benefits of VC and collaboration tools aside, Git(Hub) helps to operationalise the ideals of open science and reproducibility.
- Journals have increasingly strict requirements regarding reproducibility and data access. GH makes this easy (DOI integration, off-the-shelf licenses, etc.)
- I host all of the code for my papers on GH. I even use it to host and maintain my website.
- *Nature:* "Democratic databases: science on GitHub" (Perkel, 2016).

Git(Hub) + RStudio

Some terminology

- There's a lot of jargon in the open source software world. Here are some key terms to get you started:
    - GUI: Graphical User Interface. The point-and-click way of interacting with a program.
    - GitHub (GH): The online platform where you store your Git repositories.
    - GitHub Desktop: A GUI for Git that makes it easier to interact with your repositories. *(We will only use this as a backup in this course.)*
    - IDE: Integrated Development Environment. RStudio is an example of an IDE.
    - R: The programming language that we use in this course.
    - RStudio: The IDE that we use for R programming.

Seamless integration

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.

Seamless integration

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.

I also want to bookmark a general point that we'll revisit many times during this course:

- The tools that we're using all form part of a coherent data science ecosystem.
- Greatly reduces the cognitive overhead ("aggregation") associated with traditional workflows, where you have to juggle multiple programs and languages at the same time.

Two ways: PATs and SSH keys

- There are two ways to interact with GitHub from RStudio: Personal Access Tokens (PATs) and SSH-keys.
- PATs are a way to authenticate yourself with GitHub. They are a bit easier to set up, but are less secure than SSH-keys.
    - They rely on https authentication, roughly like adding username/password
    - Two types of PATS: fine-grained and classic
    - We'll use the classic PATs today, fine-grained are more secure but require more setup
- SSH keys are a way to identify trusted computers, without involving passwords.
    - A private key on your computer that matches a public key to put on servers, GitHub, etc.
    - Never reveal the private key, only the public
- Today I'm prioritizing a PAT approach as that is Git's latest recommendation.
    - They go back and forth on this often!
- Check the appendix for how to do get an SSH key

Setup a PAT and add to RStudio

1. Type `install.packages(c('gitcreds','usethis'))` in the R console to install the necessary packages.
2. Go to the web page: https://github.com/settings/tokens or type `usethis::create_github_token()` in the R console.
3. Click `Generate new token` and select `Generate new token (classic)`
4. Name your token something like "RStudio PAT"
5. Set it to expire on a Custom date: `04/18/2025` (so you have it for all of class, but don't forget to delete it)
6. Give it scopes into `repo`, `gist`, `workflow`, and `user`
7. Click `Generate token` and you'll see a string like: `ghp_asdfASDasdjfasdsdfaDSAF`
8. Copy to your clipboard (and leave this page open)
9. In RStudio, type `gitcreds::gitcreds_set()` and paste your token when prompted
10. Save your PAT somewhere secure as well
11. Close the page once you enter the token -- you can never see the PAT on GitHub again

Caution:

- Never share your PAT with anyone.
- Never put your PAT in code
- If your PAT is ever compromised, delete it on GitHub, generate a new one, and update your RStudio PAT

Link a GitHub repo to an RStudio Project

These slides track the class exercise for today /03-git-basics/. They walk you through a basic Git workflow.

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 link (the green "Code" button).[1]
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`.

[1] If you set up an SSH key, you would copy the SSH link instead to use the more secure protocol.

Link a GitHub repo to an RStudio Project

These slides track the class exercise for today /03-git-basics/. They walk you through a basic Git workflow.

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:

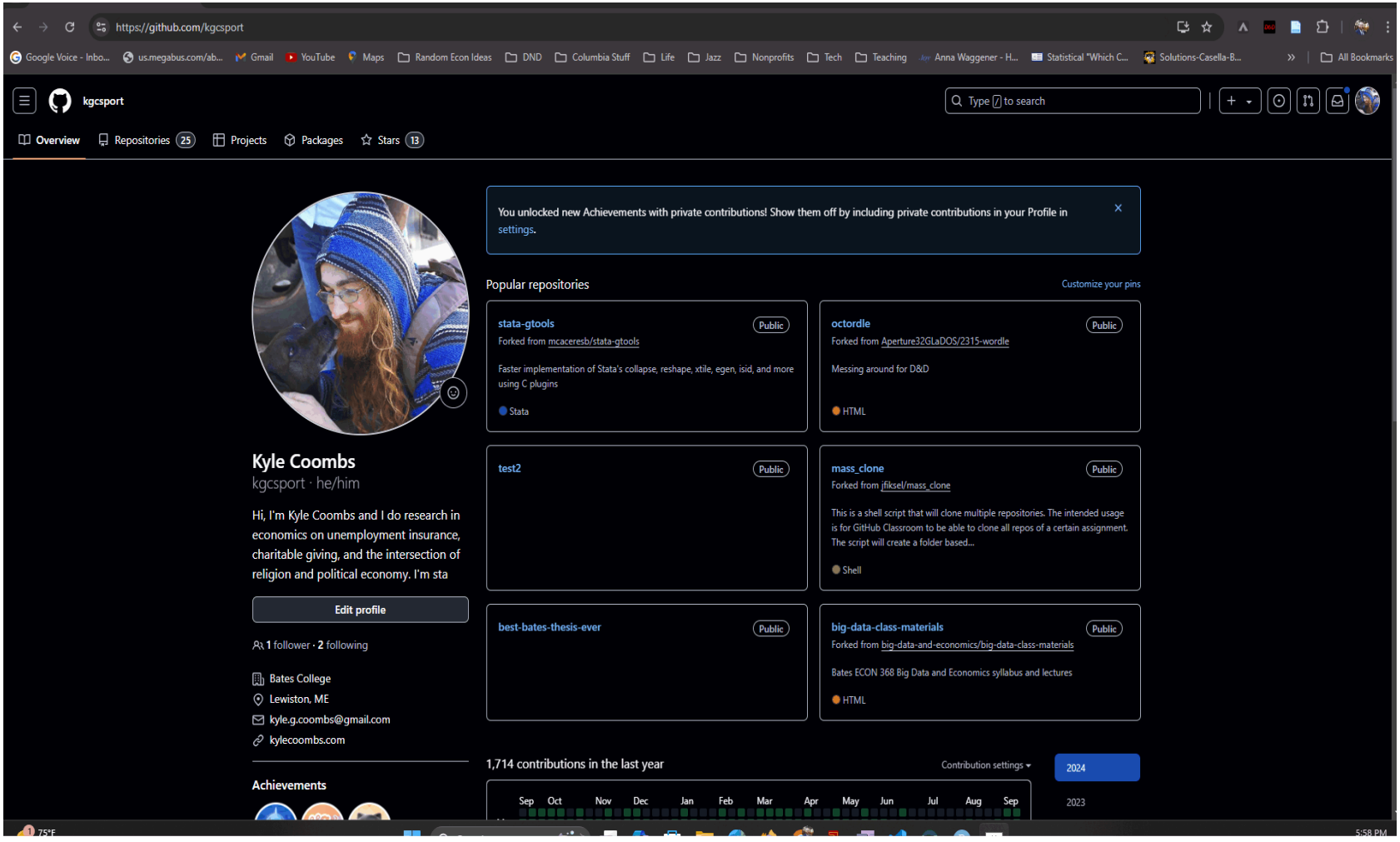
1. Create the repo on GitHub and initialize with a README.
2. Copy the HTTPS link (the green "Code" button).[1]
3. Open up RStudio.
4. Navigate to `File → New Project → Version Control → Git`.
5. Paste your copied link into the "Repository URL:" box.
6. Choose the project path ("Create project as subdirectory of:") and click `Create Project`.

Now, I want you to practice by these steps by creating your own repo on GitHub — call it "test" — and cloning it via an RStudio Project.

- See Grant's GIF walkthrough on the next slide...

[1] If you set up an SSH key, you would copy the SSH link instead to use the more secure protocol.

Link a GitHub repo to an RStudio Project



Make some local changes

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

- Click on it.
- There should already be some files in there, which we'll ignore for the moment.[1]

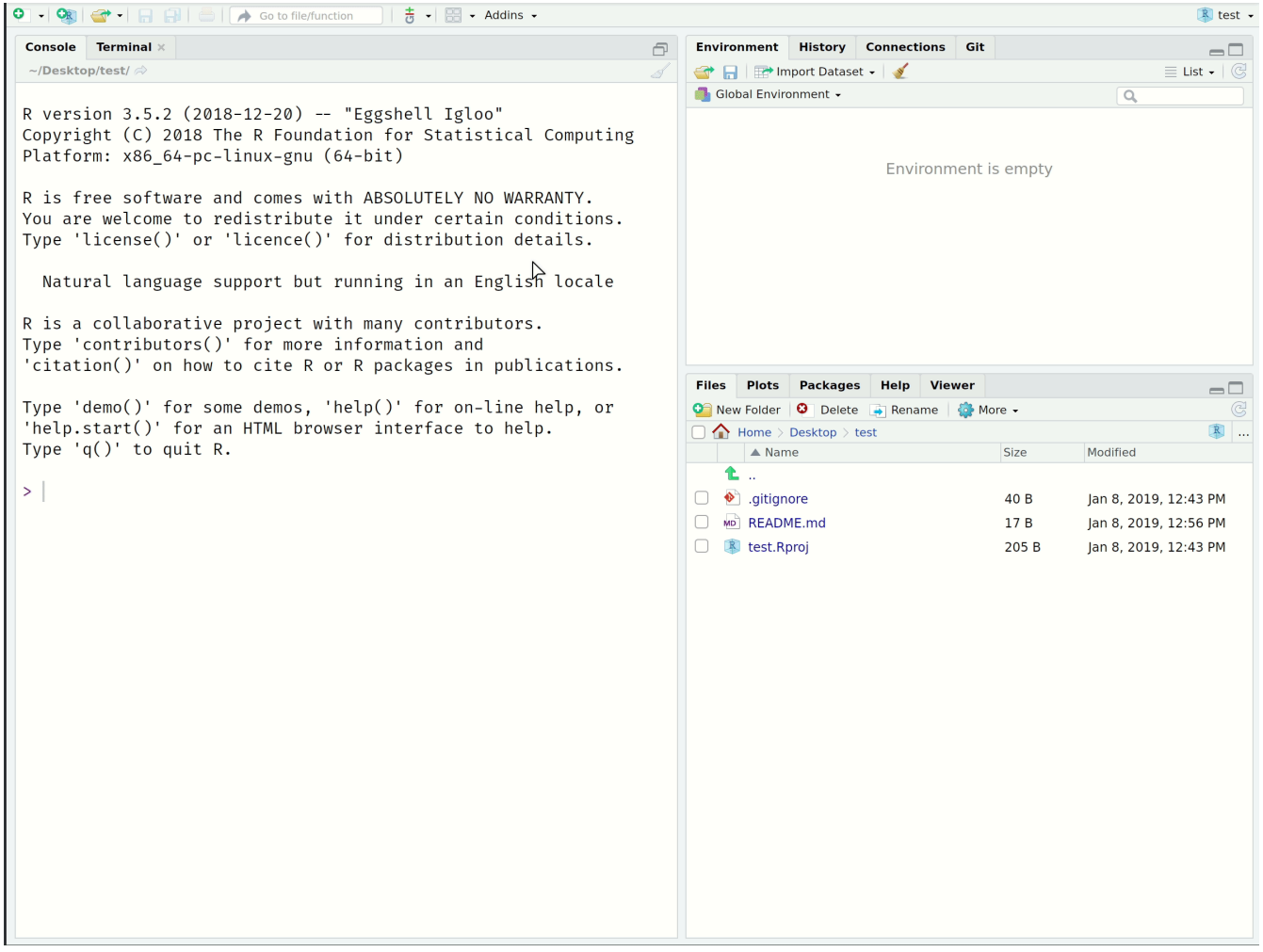
Now open up the README file (see the "Files" tab in the bottom-right panel).

- Add some text like "Hello World!" and save the README.
- Do you see any changes in the "Git" panel? Good. (Raise your hand if not.)

Again, see Grant's GIF walkthrough on the next slide...

[1] They're important, but not for the purposes of this section.

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. **Pull**
    - Get any new changes made on the GitHub repo (i.e. the upstream remote), either by your collaborators or you on another machine.
4. **Push**
    - Push any (committed) local changes to the GitHub repo

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. **Pull**
    - Get any new changes made on the GitHub repo (i.e. the upstream remote), either by your collaborators or you on another machine.
4. **Push**
    - Push any (committed) local changes to the GitHub repo

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

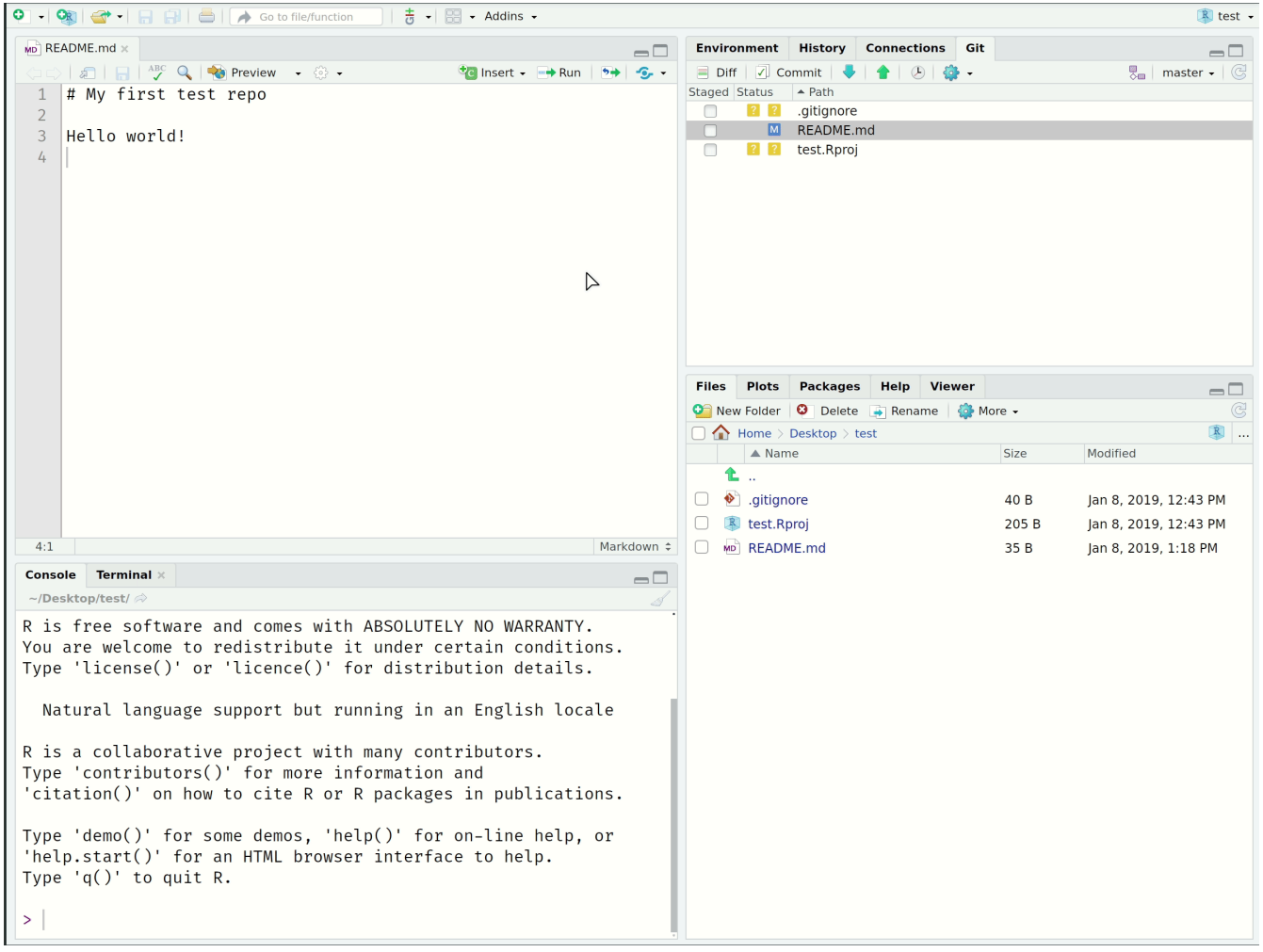
Now that you've cloned your first repo and made some local changes, it's time to learn the four main Git operations.

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. **Pull**
    - Get any new changes made on the GitHub repo (i.e. the upstream remote), either by your collaborators or you on another machine.
4. **Push**
    - Push any (committed) local changes to the GitHub repo

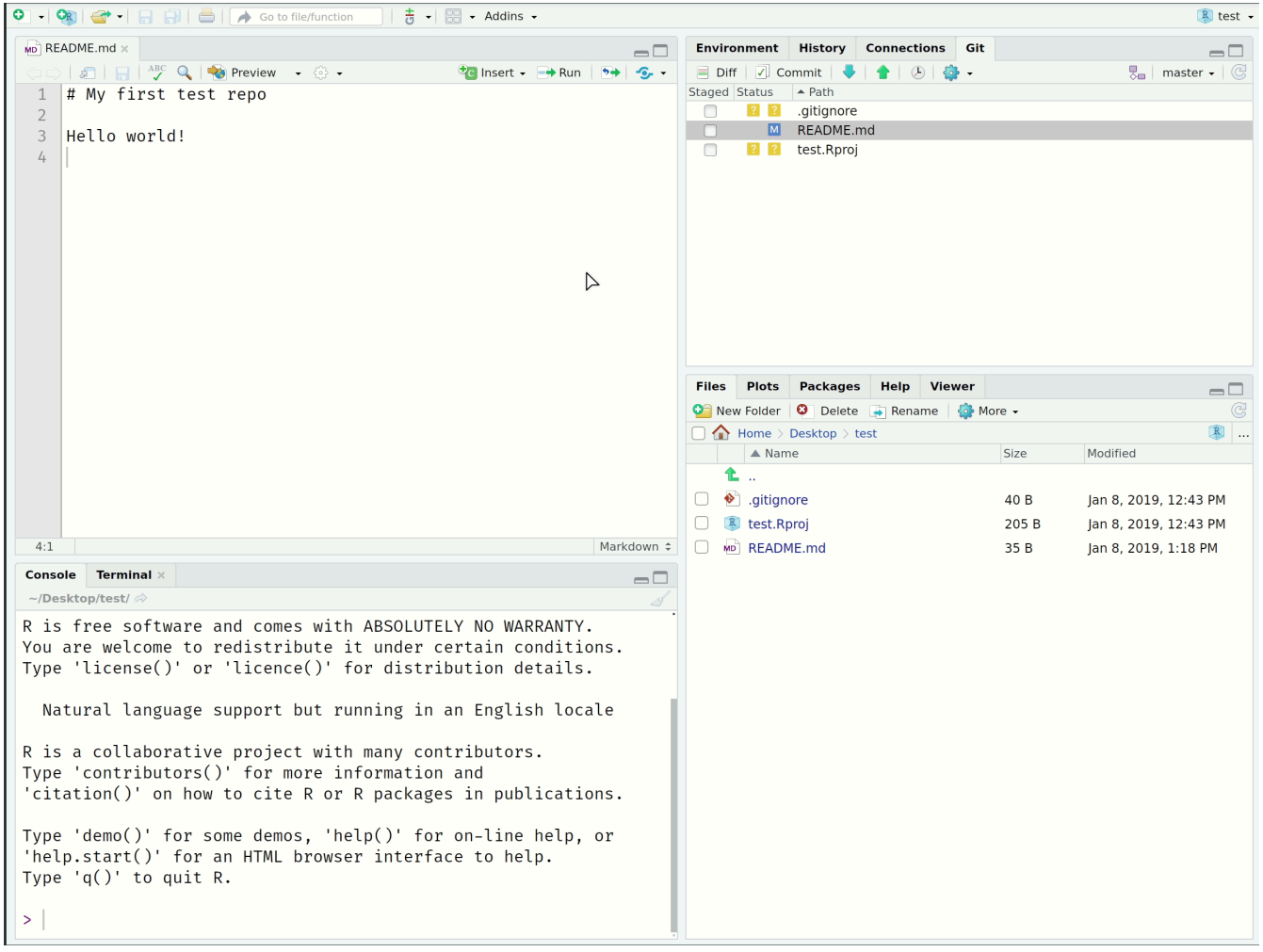
For the moment, it will be useful to group the first two operations and last two operations together. (They are often combined in practice too, although you'll soon get a sense of when and why they should be split up.)

Ready for more GIFs?

Stage and Commit

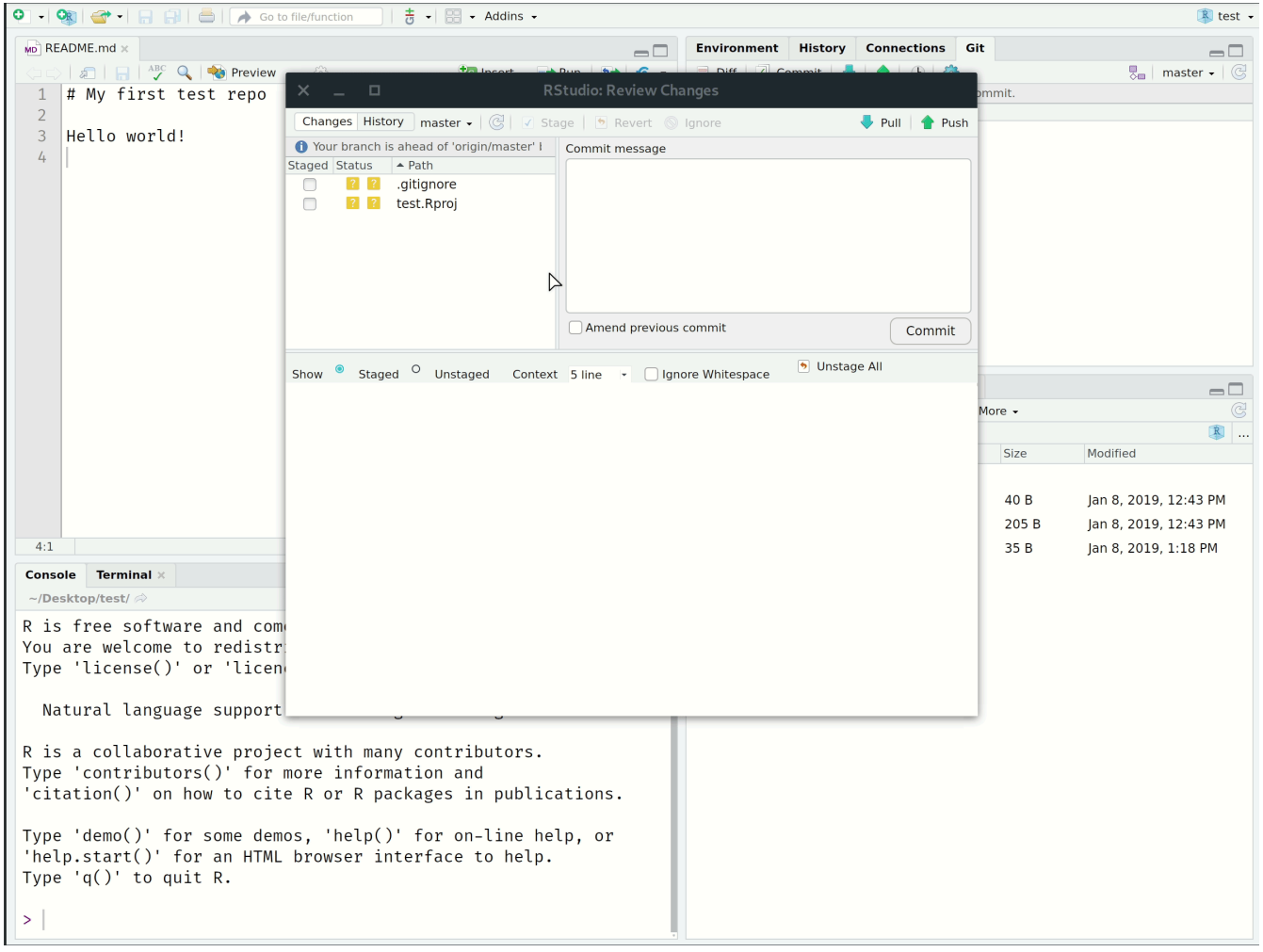


Stage and Commit

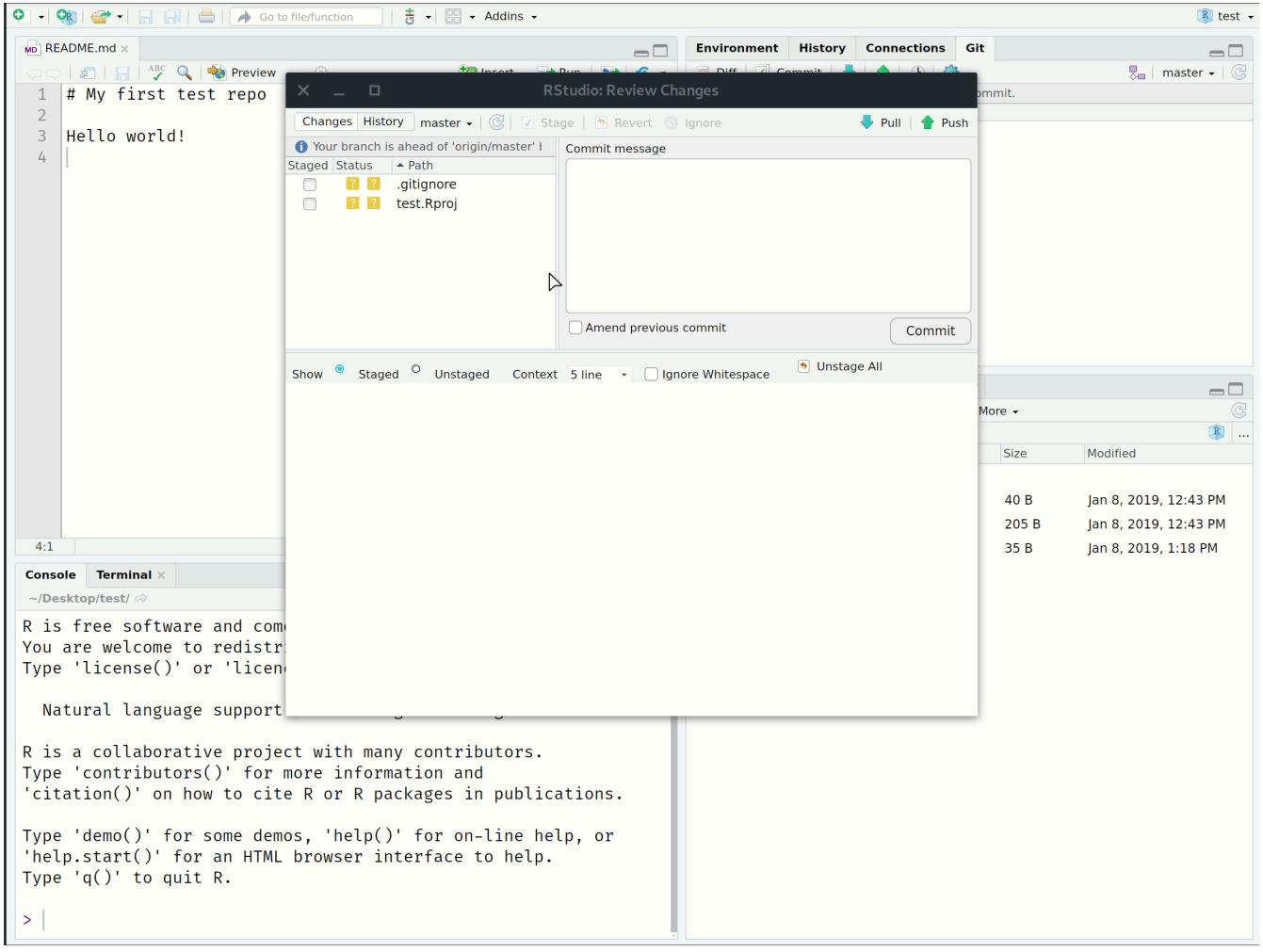


Note the helpful commit message to ourselves.

Pull then push



Pull then push



See here if you get `Error: unable to read askpass response from 'rpostback-askpass'`.

Recap

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

- Made same changes to a file and saved them locally.
- *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.

Note: 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.

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 makes you a bad person, right?

.gitignore

A .gitignore file tells Git what to — *wait for it* — 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.

I typically add compiled datasets to my .gitignore in the early stages of a project.

- Reduces redundant version control history, so I only save the raw data and the code used to compile it
- Simple to remove from my .gitignore once the project is being finalised (e.g. paper is being submitted).

.gitignore (cont.)

You can create a .gitignore file in multiple ways.

- 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".)

Once the .gitignore file is created, simply add in lines of text corresponding to the files that should be ignored.

- 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`

Let's ignore some file types!

- Open up your `.gitignore` file in RStudio.
- Add the following line of text: `*.csv` to your `.gitignore`
- Save the file and run the following code in your R console:

```
download.file('https://tinyurl.com/econ368ignore',destfile='my_data.csv')
```

- Check your git tab in Rstudio

1. The `.gitignore` file should be in need of staging
2. The `my_data.csv` file should not appear despite being in your working directory

- Why not?

Let's ignore some file types!

- Open up your `.gitignore` file in RStudio.
- Add the following line of text: `*.csv` to your `.gitignore`
- Save the file and run the following code in your R console:

```
download.file('https://tinyurl.com/econ368ignore',destfile='my_data.csv')
```

- Check your git tab in Rstudio

1. The `.gitignore` file should be in need of staging
2. The `my_data.csv` file should not appear despite being in your working directory

- Why not?

Because we told Git to ignore all CSV files in the `.gitignore` file!

Merge conflicts

Collaboration time

Turn to the person next to you. You are now partners. (Congratulations.)

- P1: Invite P2 to join you as a collaborator on the "test" GitHub repo that you created earlier. (See the *Settings* tab of your repo.)
- P2: Clone P1's repo to your local machine.[1] Make some edits to the README (e.g. delete lines of text and add your own). Stage, commit and push these changes.
- P1: Make your own changes to the README on your local machine. Stage, commit, pull from the GitHub repo, and then try to push them.

[1] Change into a new directory first or give it a different name to avoid conflicts with your own "test" repo. Don't worry, Git tracking will still work if you change the repo name locally.

Collaboration time

Turn to the person next to you. You are now partners. (Congratulations.)

- P1: Invite P2 to join you as a collaborator on the "test" GitHub repo that you created earlier. (See the *Settings* tab of your repo.)
- P2: Clone P1's repo to your local machine.[1] Make some edits to the README (e.g. delete lines of text and add your own). Stage, commit and push these changes.
- P1: Make your own changes to the README on your local machine. Stage, commit, pull from the GitHub repo, and then try to push them.

Did P1 encounter a `merge conflict` error?

- Good, that's what we were trying to trigger.
- Now, let's learn how to fix them.

[1] Change into a new directory first or give it a different name to avoid conflicts with your own "test" repo. Don't worry, Git tracking will still work if you change the repo name locally.

Did you not get a merge conflict?

- Some of you may have encountered a divergent branch:

```
Hint: You have divergent branches and need to specify how to reconcile them.
Hint: You can do so by running one of the following commands sometime before
Hint: your next pull:
Hint:
Hint:   git config pull.rebase false  # merge
Hint:   git config pull.rebase true   # rebase
Hint:   git config pull.ff only       # fast-forward only
Hint:
```

- Per an update in 2022, you should not be getting this message as a fatal error
  - If you got this error, you likely do not have the latest version of Git installed as requested. Fix this.
- Note that GitHub issues and pull requests were used by professionals to improve the product over time

Divergent branches

- We'll get into branches below, but essentially this is because `git pull` does two things:
  - `git fetch` which downloads the latest changes from the remote repo
  - `git merge` which merges the changes into your local repo
- For your purposes, I recommend you enter the following in your Terminal (next to Console in RStudio):

```
git config --global pull.ff merge # fast-forward and merge
```

- This will change your `.gitconfig` file, which sets the settings of git and can be accessed with `usethis::edit_git_config()` in RStudio
- Solution pulled from GitHub Desktop issues
- As always, get more advice from Jenny Bryan

Merge conflicts

Let's confirm what's going on. Type this into your terminal:

```
$ git status
```

As part of the response, you should see something like:

```
Unmerged paths:
  (use "git add <file>..." to mark resolution)

   * both modified:   README.md
```

Git is protecting P1 by refusing the merge. It wants to make sure that you don't accidentally overwrite all of your changes by pulling P2's version of the README.

- In this case, the source of the problem was obvious. Once we start working on bigger projects, however, `git status` can provide a helpful summary to see which files are in conflict.

Merge conflicts (cont.)

Okay, let's see what's happening here by opening up the README file. RStudio is a good choice, although your preferred text editor is fine.[1]

You should see something like:

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

[1] Other good choices are VS Code or Atom, which both support native Git(Hub) integration. You can set your preferred default editor with `$ git config --global core.editor "PREFERRED_EDITOR"`.

Merge conflicts (cont.)

What do these symbols mean?

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

Merge conflicts (cont.)

What do these symbols mean?

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

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

Merge conflicts (cont.)

What do these symbols mean?

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

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

Merge conflicts (cont.)

What do these symbols mean?

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

- `<<<<<<< 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.

Merge conflicts (cont.)

Fixing these conflicts is a simple 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.

Merge conflicts (cont.)

Fixing these conflicts is a simple 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

- P1 gets to decide what to keep because they fixed the merge conflict.
- OTOH, the full commit history is preserved, so P2 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**. We'll get there next.

Advice

- Merge conflicts can create a bit of a headache if you're not careful.
- Often, you can avoid merge conflicts by pulling from the upstream repo before you start working on your local copy of code.
    - This is especially true if you are working on a project with multiple collaborators.
    - Before you start editing code, always pull from the upstream repo to make sure that you have the latest version.
- If you do encounter a merge conflict, don't panic. It's not the end of the world.
    - Git is designed to preserve your work and the work of your collaborators.
    - You can always revert to a previous commit if you need to.
- Jenny Bryan has a great section on Git Workflows to help you use Git productively

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, I want to go over Git shell commands so that you can internalise the basics.

- The shell is more powerful and flexible. Does some things that the RStudio Git GUI can't.
- Potentially more appropriate for projects that aren't primarily based in R. (Although, no real harm in using RStudio Projects to clone a non-R repo.)
- Also, I don't want to screen record more

Main Git shell commands

Clone a repo.

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

See the commit history (hit spacebar to scroll down or q to exit).

```
$ git log
```

What has changed?

```
$ git status
```

Main Git shell commands (cont.)

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 .
  ```

Main Git shell commands (cont.)

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
```

Branches and forking

What are branches and why use them?

Branches are one of Git's coolest features.

- Allow you to take a snapshot of your existing repo and try out a whole new idea *without affecting* your main branch.[1]
- Only once you (and your collaborators) are 100% satisfied, would you merge it back into the main branch.[2]
    - 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 — and should! — 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.

[1] Github used to call the main branch "master", but has now switched to "main."

[2] You can actually have branches of branches (of branches). But let's not get ahead of ourselves.

What are branches and why use them?

Branches are one of Git's coolest features.

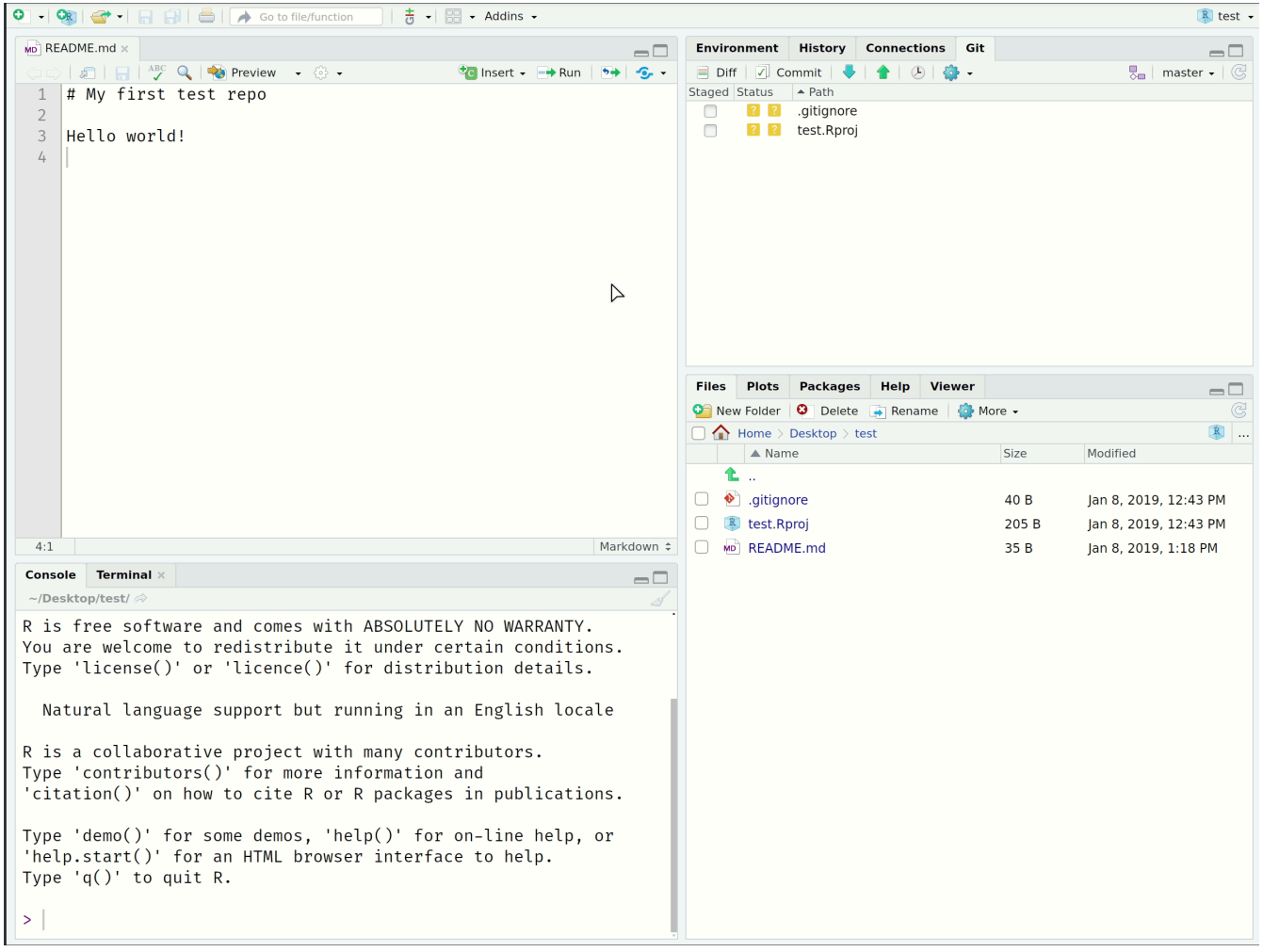
- Allow you to take a snapshot of your existing repo and try out a whole new idea *without affecting* your main branch.[1]
- Only once you (and your collaborators) are 100% satisfied, would you merge it back into the main branch.[2]
    - This is how most new features in modern software and apps are developed.
    - It is also how bugs are caught and fixed.
    - But researchers can easily — and should! — 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.

I use branches all the time for my own research projects.

[1] Github used to call the main branch "master", but has now switched to "main."

[2] You can actually have branches of branches (of branches). But let's not get ahead of ourselves.

Create a new branch in RStudio



Branch shell commands

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 --delete NAME-OF-YOUR-FAILED-BRANCH
```

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 main branch: `$ git checkout main`
- 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 practice this now in class...

Your first pull request

You know that "new-idea" branch we just created a few slides back? Switch over to it if you haven't already.

- Remember: `$ git checkout new-idea` (or just click on the branches tab in RStudio)

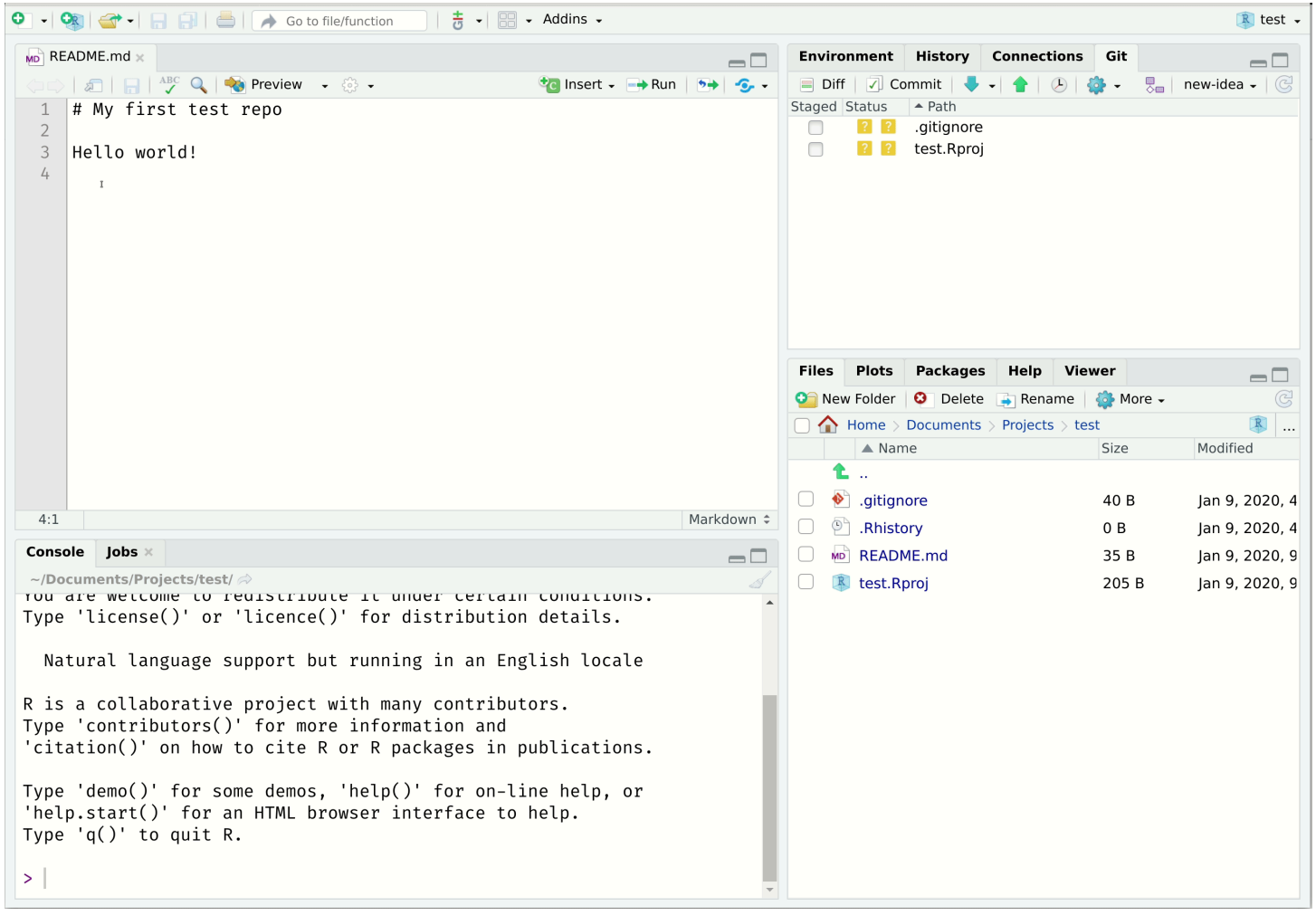
Make some local changes and then commit + push them to GitHub.

- The changes themselves don't really matter. Add text to the README, add some new files, whatever.

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

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.

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.
- Try this now. Use one of the class repos if you can't think of anyone else's.

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.

Forks (cont.)

Creating forks is super easy as we've just seen. However, maintaining them involves some more leg work if you want to stay up to date with the original repo.

- GitHub: "Syncing a fork"
- OTOH, this isn't going to be an issue for completed projects. E.g. Forking the repo that contains the code and data of a published paper.

Forks (cont.)

Creating forks is super easy as we've just seen. However, maintaining them involves some more leg work if you want to stay up to date with the original repo.

- GitHub: "Syncing a fork"
- OTOH, this isn't going to be an issue for completed projects. E.g. Forking the repo that contains the code and data of a published paper.

Open Source Software contribution

Lots of Open Source Software (OSS) projects are hosted on GitHub. They rely on forks, branches, etc. to manage contributions from the community.

I will give anyone who successfully makes a contribution to an OSS project during the semester a bonus 2.5% on their final grade. (Link me to the PR and I'll verify.)

- Grades aside, I want to encourage you to start thinking about contributing to software projects in general.
- Seriously, it can be something as simple as correcting typos or language. Many great programmers and data scientists are not English first-language speakers. Helping to improve package documentation is a small way to say thanks. (More here.)

Other tips

README

README files are special in GitHub because they act as repo landing pages.

- 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).

README files can also be added to the *sub-directories* of a repo, where they will act as a landing pages too.

- Particularly useful for bigger projects. Say, where you are using multiple programming languages (e.g. here), or want to add more detail about a dataset (e.g. here).

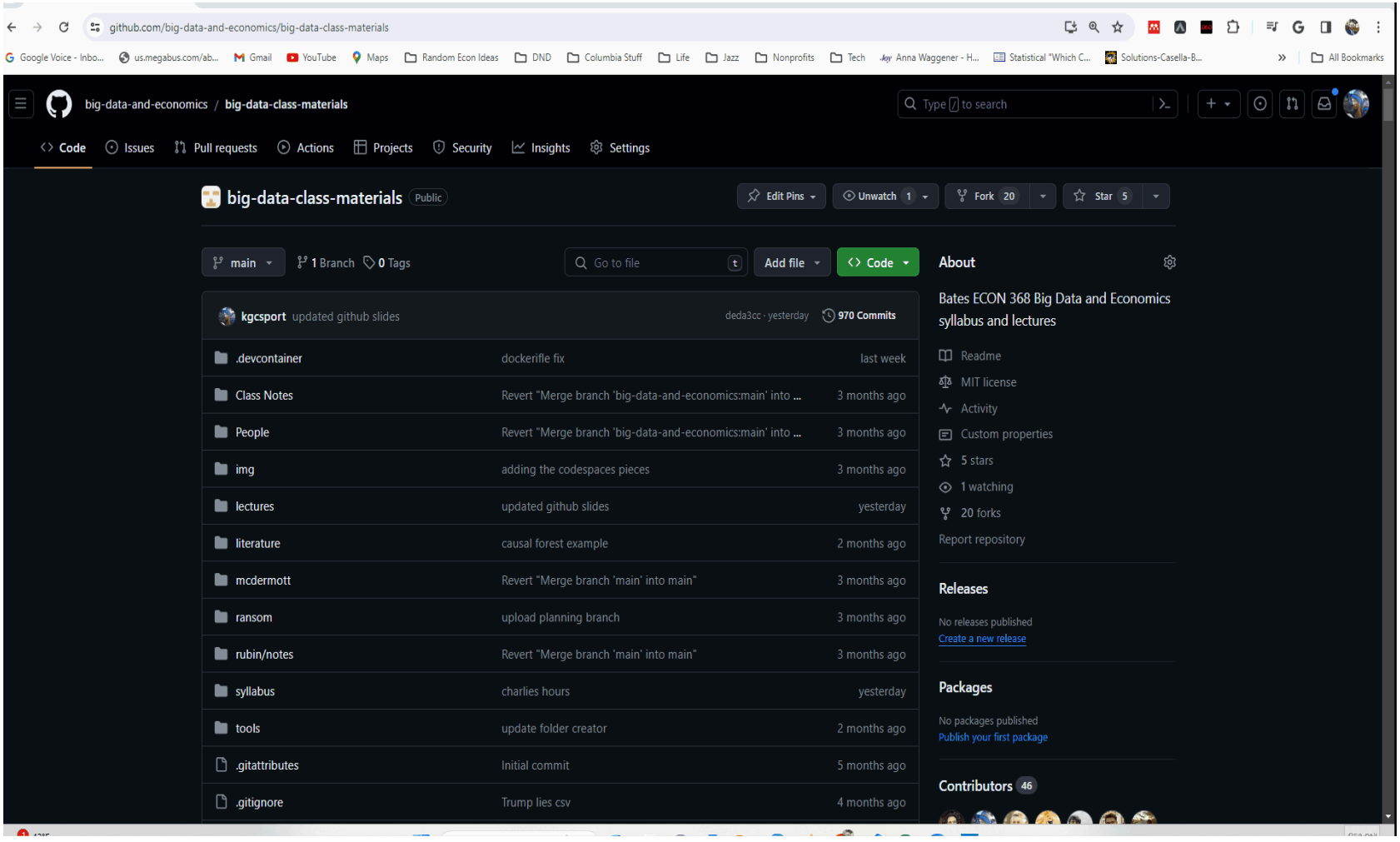
READMEs should be written in Markdown, which GH automatically renders.

- We'll learn more about Markdown (and its close relation, R Markdown) during the course of our homework assignments.

GitHub Codespaces

- GitHub Codespaces is a new feature that allows you to code directly in the browser
- All of you have access to this as members of the class GitHub organization and/or if you registered with GitHub Education
- You can access it by clicking the `Code` dropdown menu > `Codespaces` tab > `Create codespace on main` (or whatever branch you prefer)
- It is a great tool to fiddle with a repository in a browser without having to install anything on your local machine
- It is also a great tool to test drive a coding project in a controlled environment
- In this class, there is a port to a server version of RStudio under the `Ports` tab in the bottom panel
    - The port is labeled `Rstudio`
    - The username/password are "rstudio" and "rstudio"
    - I show a gif of this use case on the next slide

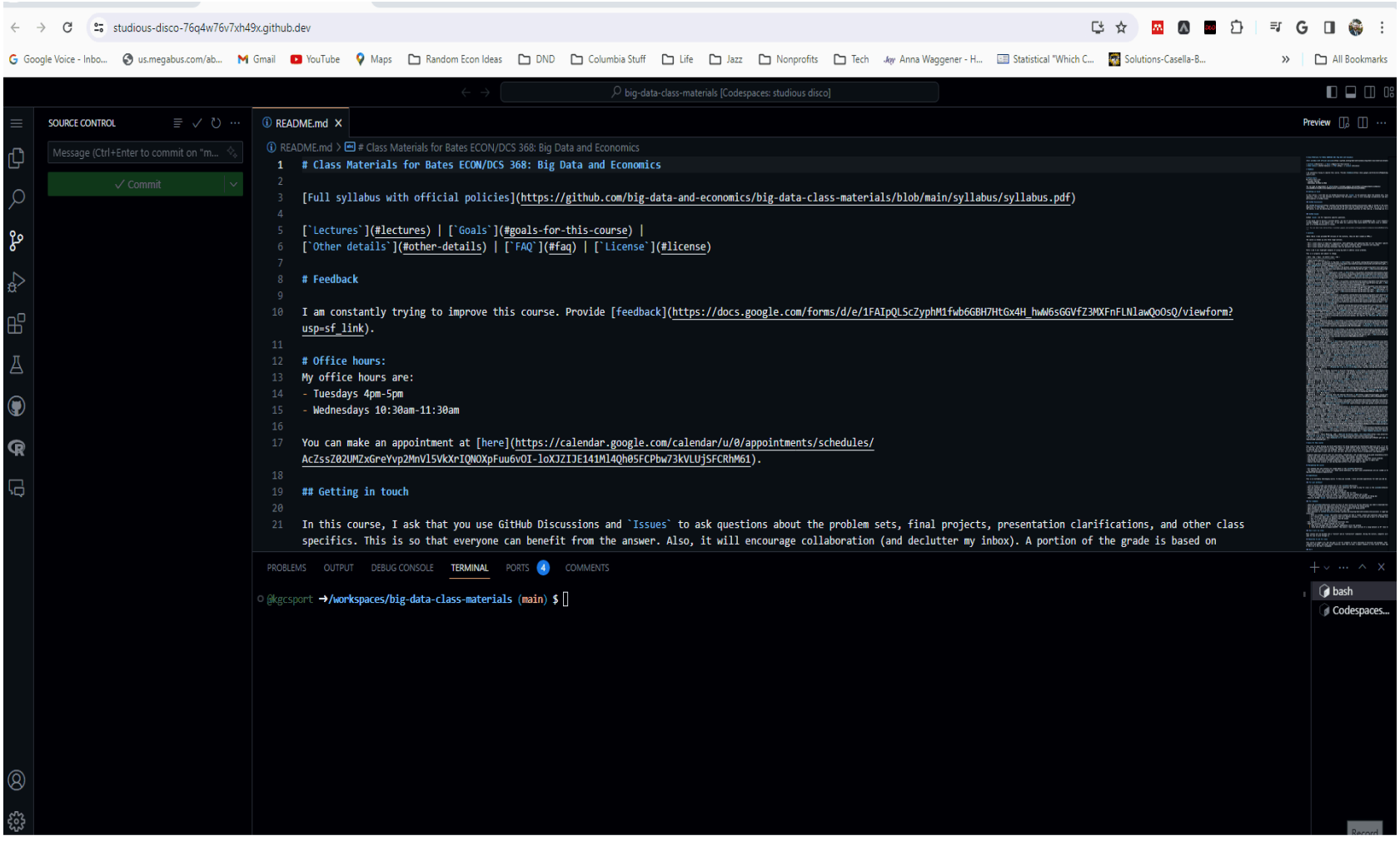
GitHub Codespace/RStudio launch



Git source control in VS Code

- VS Code has built-in Git source control integration.
- This is what you will use in GitHub Codespaces
- The "Git" vocabulary is the same as what we've learned in RStudio, but the UI is different.
- You will need to `git fetch` to pull from the upstream repository (i.e. GitHub)
- It is built a little smarter and will autmatically pull, then push when you select "sync" from the UI
- This is not exactly "best practice," but it does make it easier to get started with Git
- I show a `pull` and `sync` in the GIF on the next slide, but the documentation is linked above
- ALWAYS ALWAYS ALWAYS make sure you use source control to push/pull your changes after coding in the RStudio port

Git source control gifs



GitHub Issues

GitHub Issues are another great way to interact with your collaborators and/or package maintainers.

- If you spot any problems with these lecture notes, please file an issue here! (Keep in mind that is public!)

Summary

Recipe (shell commands in yellow)

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 yellow)

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.

Appendix

Creating an SSH-key

Before we get started, raise your hand if you did not successfully create an SSH key on your local machine. You will need one today.

SSH-Keys

- A key has a type of encryption, two examples:
  - RSA (Rivest-Shamir-Adleman)
  - ED25519 (Edwards-curve Digital Signature Algorithm)
- We will use ED25519, which is smaller, more secure and faster than RSA
  - GitHub also recommends it
- We will generate a key, then add the public key to GitHub SSH keys.
  - I will show you the "hard" way first, then an "easier" way second (guess which we'll do?)
- You can add a passphrase to your SSH-key, but you will need to remember that password every time you push/pull. (I often don't use a passphrase for my SSH-keys unless it is for a highly sensitive use case.)

The hard way -- ssh-keygen

- You can generate an SSH key via the terminal with the following command:

```
ssh-keygen -t ed25519  -C "YOUR-EMAIL-ADDRESS OR OTHER COMMENT"
```

- You will be prompted to enter a file in which to save the key. Just hit enter to accept the default location.
- You will then be prompted to enter a passphrase. You can either enter a passphrase or leave it blank. (If you leave it blank, you will not be prompted for a passphrase when you use the key.)
- You should see something like this:

```
Generating public/private ed25519 key pair.
Enter file in which to save the key (/Users/you/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/you/.ssh/id_ed25519
Your public key has been saved in /Users/you/.ssh/id_ed25519.pub.
```

Add SSH-key to ssh-agent, GitHub

- Then you'll need to add the key to the ssh-agent:

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519
```

  Note: This is OS-specific, see instructions here

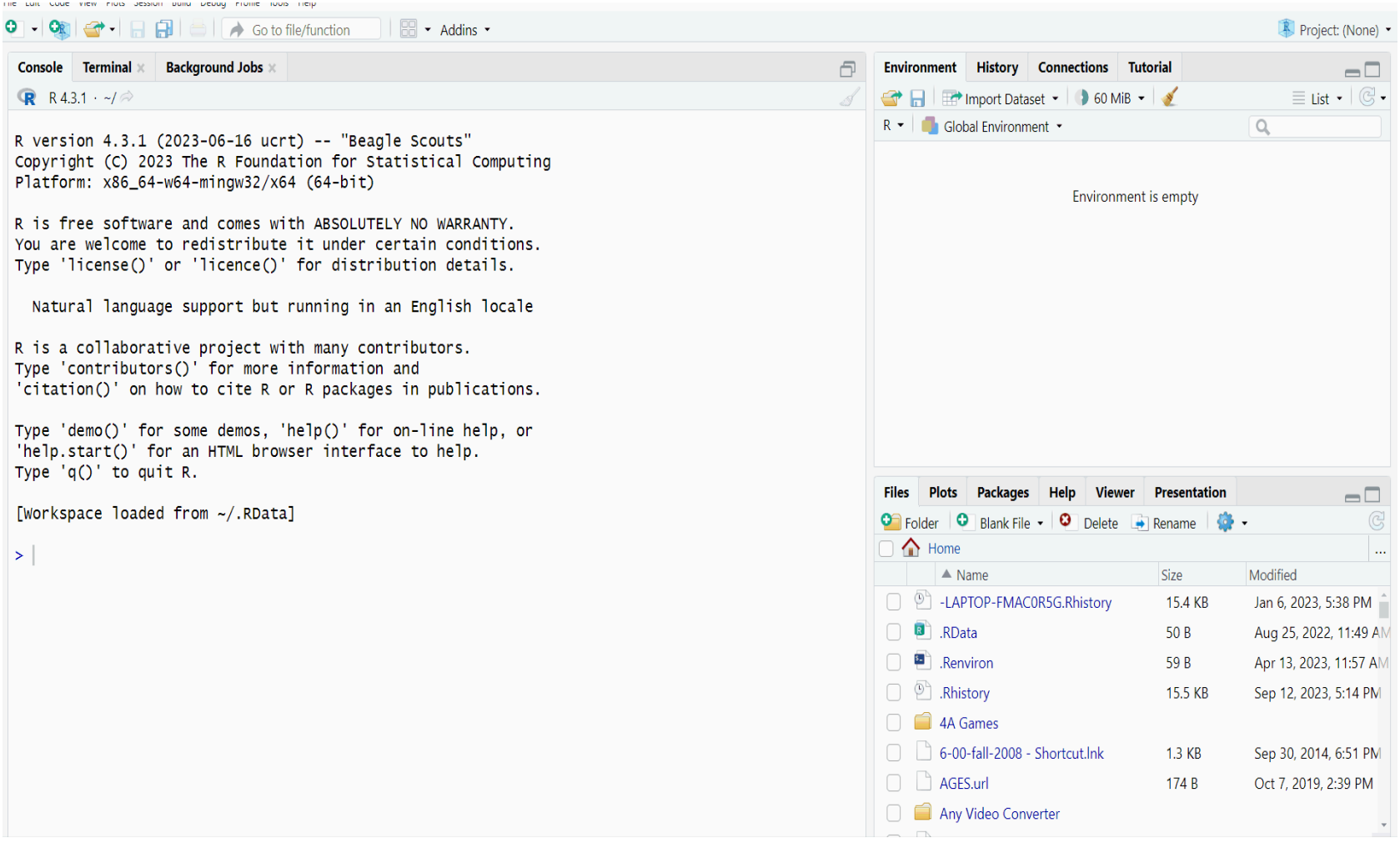
- Last you'll navigate to your ssh-key in your file system and copy the contents of the public key (id_ed25519.pub) to your clipboard.

- Then you'll go to your GitHub account settings and add the public key to your account under SSH keys.

Point-and-click with RStudio

- You can also point-and-click with RStudio: `Tools → Global Options → Git/SVN → Create RSA Key ...`
    - Guess which we're gonna do today?
- If you already have an SSH key on your local machine, then you can skip this step.
- Instead, RStudio will already see your key and you can click "View public key" to copy it to your clipboard and add to GitHub

A gif how-to


Go to file/function
Addins
Project: (None)
Console
Terminal
Background Jobs
R 4.3.1 · ~/
R version 4.3.1 (2023-06-16 ucrt) -- "Beagle Scouts"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
[Workspace loaded from ~/.RData]
>
Environment
History
Connections
Tutorial
Import Dataset
60 MiB
List
R
Global Environment
Environment is empty
Files
Plots
Packages
Help
Viewer
Presentation
Folder
Blank File
Delete
Rename
Home
Name
Size
Modified
-LAPTOP-FMAC0R5G.Rhistory 15.4 KB Jan 6, 2023, 5:38 PM
.RData 50 B Aug 25, 2022, 11:49 AM
.Renviron 59 B Apr 13, 2023, 11:57 AM
.Rhistory 15.5 KB Sep 12, 2023, 5:14 PM
4A Games
6-00-fall-2008 - Shortcut.lnk 1.3 KB Sep 30, 2014, 6:51 PM
AGES.url 174 B Oct 7, 2019, 2:39 PM
Any Video Converter

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. (More info here and here.)

- For Linux and MacOS, that ending is "LF"
- For Windows, that ending is "CRLF" (of course it is...)

Solution

Open up the shell and enter

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

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

FAQ

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!"

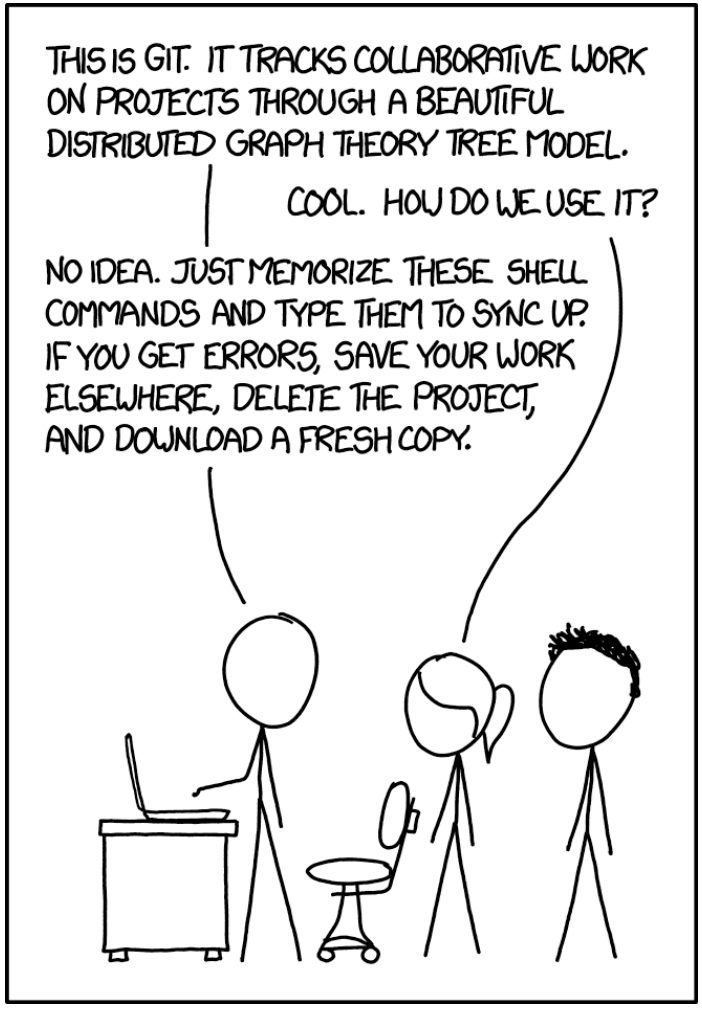
- Seriously: http://ohshitgit.com/.

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.

FAQ (cont.)


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.